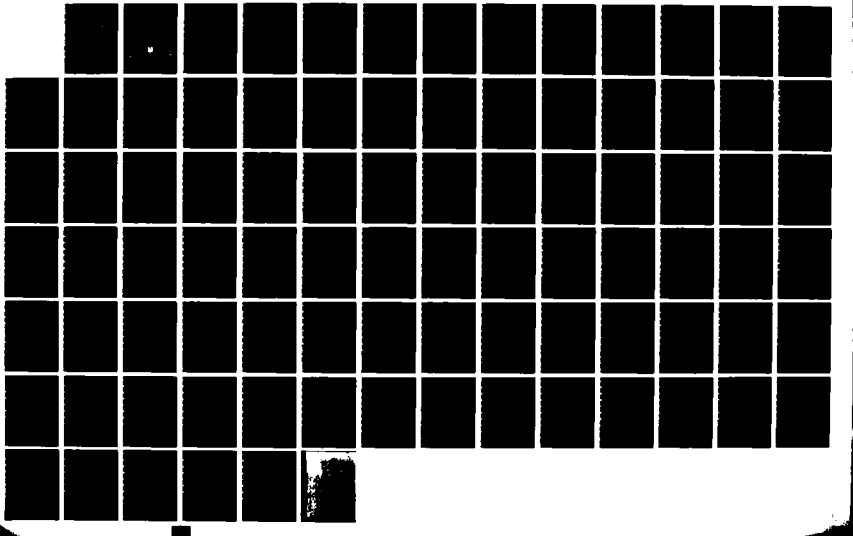
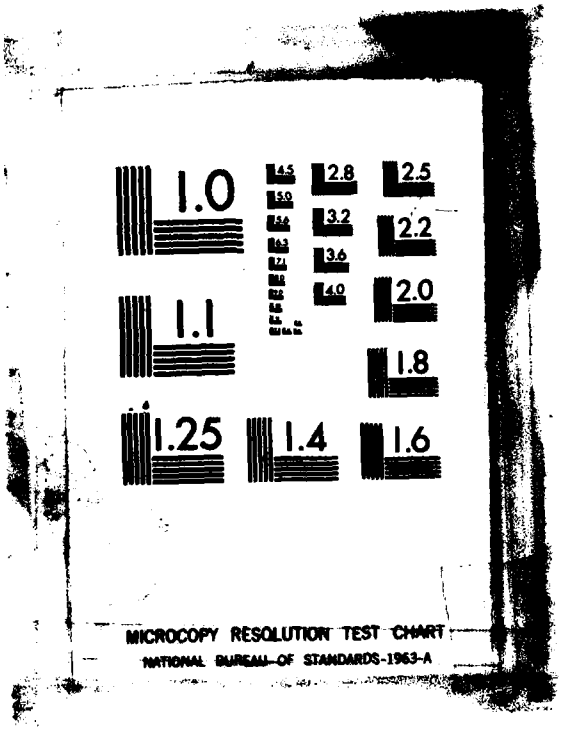
AD-A140 884 A STUDY OF THE FEASIBILITY OF DUPLICATING JAMPS APPLICATIONS SOFTWARE IN THE ADA PROGRAMMING LANGUAGE (U) MITRE CORP BEDFORD MA R G HOWE APR 84 MTR-9167 1/1

UNCLASSIFIED ESD-TR-84-160 F19628-84-C-0001 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

ESD-TR-84-160 MTR-9167

AD-A140 884

A STUDY OF THE FEASIBILITY OF DUPLICATING JAMPS APPLICATIONS SOFTWARE IN THE ADA PROGRAMMING LANGUAGE

By
R. G. HOWE

APRIL 1984

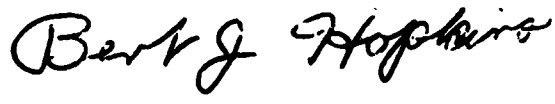
Prepared for
DEPUTY COMMANDER FOR TACTICAL SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

Project No. 4100
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-84-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

JAMES C. GRAVES, 1Lt, USAF
TADIL J Systems Engineer
Tactical Communication Systems



BERT J. HOPKINS, GM-13
Chief, Special Projects Division
Tactical Communication Systems

FOR THE COMMANDER

RICHARD M. DEMILIA
Asst System Program Director
Tactical Communication Systems Program Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MTR-9167 ESD-TR-84-160	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Burlington Road Bedford, MA 01730	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Deputy Commander for Tactical Systems 8b. OFFICE SYMBOL (If applicable) TCSR	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-84-C-0001
8c. ADDRESS (City, State and ZIP Code) Electronic Systems Division, AFSC Hanscom AFB, MA 0	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. 4100 / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) A STUDY OF THE FEASIBILITY OF DUPLICATING	
12. PERSONAL AUTHOR(S) R. G. Howe	
13a. TYPE OF REPORT Final Report 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1984 April 15. PAGE COUNT 86
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB. GR.		
			Ada	Cocomo Cost Model
			Ada Run-Time Environments	Ada Feasibility
			Ada Software Cost Estimation	JAMPS

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document is a feasibility study of reimplementing the JAMPS applications software using the Ada programming language. Existing JAMPS software is written in "C" language; reimplementation is under consideration to promote the reusability of the JAMPS software and decrease JAMPS life cycle costs. Ada software development tools for the MC68000 now exist in rudimentary form, but, due to the inadequacy of run time environments and the lack of validated compilers, these tools are inadequate for duplicating JAMPS software at this time. However, the tools are expected to improve sufficiently that reimplementation in Ada might reasonably begin in FY85. Cost estimates result in a $4.5M pricetag; manpower estimates and schedules are also included.

(JINTACCS AUTOMATED MESSAGE PROCESSING SYSTEM)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Susan R. Gilbert	22b. TELEPHONE NUMBER (Include Area Code) (617) 271-8088 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

11. (Con't.) JAMPS APPLICATIONS SOFTWARE IN THE ADA PROGRAMMING LANGUAGE.

EXECUTIVE SUMMARY

The existing JAMPS software, written in the "C" language, was developed under severe budgetary constraints and without any consideration of provisions for software reusability. Consequently, it may be difficult to incorporate JAMPS software as an off-the-shelf package in other Air Force systems. Reimplementation in Ada is under consideration to promote the reusability of JAMPS software.

Ada software development tools (for the MC68000) currently exist in rudimentary form, but are not considered adequate for duplicating JAMPS software at this time. However, the tools are expected to improve sufficiently that reimplementaton in Ada might reasonably begin in FY85.

If, after appropriate Ada programming support tools become fully available, there is serious interest in reusing JAMPS software in several other Air Force acquisition programs, then it would be reasonable to reconsider the possibility of duplicating JAMPS software in Ada.

Once a contract has been awarded, it will take an estimated two years to duplicate JAMPS software in Ada.

The total cost to duplicate JAMPS software in Ada is estimated to be $4.5M (1983 dollars). An upper bound for the total cost of reimplementation in Ada, based on some pessimistic assumptions, is $6.3M.

ACKNOWLEDGMENTS

The author wishes to thank the following individuals for their generous support and ideas throughout the preparation of this report.

Mr. P. R. Bigelow
Mr. W. D. Brentano
Mr. D. P. Crowson
Mr. E. C. Grund
Mr. W. E. Miller, Jr.
Mr. W. G. Neumann
Mr. C. D. Poindexter

The author is especially grateful to Mr. D. J. Criscione who recoded a representative example of JAMPS "C" code using Ada, and to Ms. P. L. Mintz who provided the JAMPS sizing data included in this document.

The document has been prepared by The MITRE Corporation under Project 4100, Contract F19628-84-C-0001. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1

INTRODUCTION

1.1 BACKGROUND

JINTACCS is an acronym for the Joint Interoperability Tactical Command and Control Systems. The JINTACCS Automated Message Processing System, JAMPS, "is a portable set of computer-controlled equipment for assisting a group of collocated operators in composing and exchanging JINTACCS messages. It provides each operator with a work station for composing messages and interface equipment for sending and receiving messages over a long-haul communications link, as well as for sending messages to and receiving messages from other operators in the local group."

Prototype versions of JAMPS have been designed and built by The MITRE Corporation. JAMPS has undergone compatibility and interoperability testing and operational effectiveness demonstrations (OED) during military exercises. As a result of these exercises, it was decided that the principal processing component, the DEC 11/23, should be replaced with a Motorola MC68000-based computer and a faster disk to overcome response time problems and generally improve system performance. The JAMPS software, written in the "C" language and executed under the UNIX* operating system, is being transported to the MC68000 at this time.

Approximately a dozen Air Force organizations have expressed interest in reusing JAMPS software (but not necessarily the hardware) in other military systems. Inasmuch as the "C" language is not approved for use in DoD acquisition programs and because the use of Ada will soon become mandatory in many types of defense systems, the Electronic Systems Division of USAF has asked MITRE to investigate the feasibility of duplicating JAMPS software in the Ada language. This document, prepared in response to ESD's request, includes estimates for the time, money, and manpower required to rewrite JAMPS software in Ada. It also presents the advantages and disadvantages of undertaking such an effort. This investigation was jointly funded by Project 4100 (JAMPS) and Project 572D (Ada transition planning associated with the Computer Resources Management Program PE64740F) and consequently is more thorough than might otherwise be expected if sponsored by Project 4100 alone.

*UNIX is a trademark of Bell Laboratories.

1.2 JAMPS HARDWARE CONFIGURATION

There are two types of JAMPS hardware:

1. Work station

2. Information distribution network

The work station hardware to which the JAMPS system is now transported is for operational demonstration of the JAMPS concept. It will provide better response time performance and greater storage capacity than the demonstration hardware used in previous operational tests.

The upgraded work station includes the following hardware elements:

- o MC68000 computer, 0.5 Mbyte main memory
- o Display
- o Keyboard
- o Storage Device (40 or 70 megabyte Winchester disk drive)
- o Printer
- o Q-bus interface hardware
- o DEC DLVII-E asynchronous line interface
- o DEC DLVII-J four-part RS-232 multiplexer
- o Floppy disk drive; 8-inch, .1 Mbyte floppy disks
- o Diagnostic firmware and control panel

Figures 1 and 2 depict the hardware configuration.

1.3 FUNDAMENTAL CONCEPTS OF THE EXISTING SOFTWARE ARCHITECTURE FOR JAMPS

All valid JINTACCS message types and formats are under configuration control by the Army. Information pertaining to JINTACCS message types and formats is being supplied by the Army on a magnetic tape (i.e., the "JINTACCS Tape") to the Data Processing Center at Langley AFB, Tactical Air Command (TAC). At the center

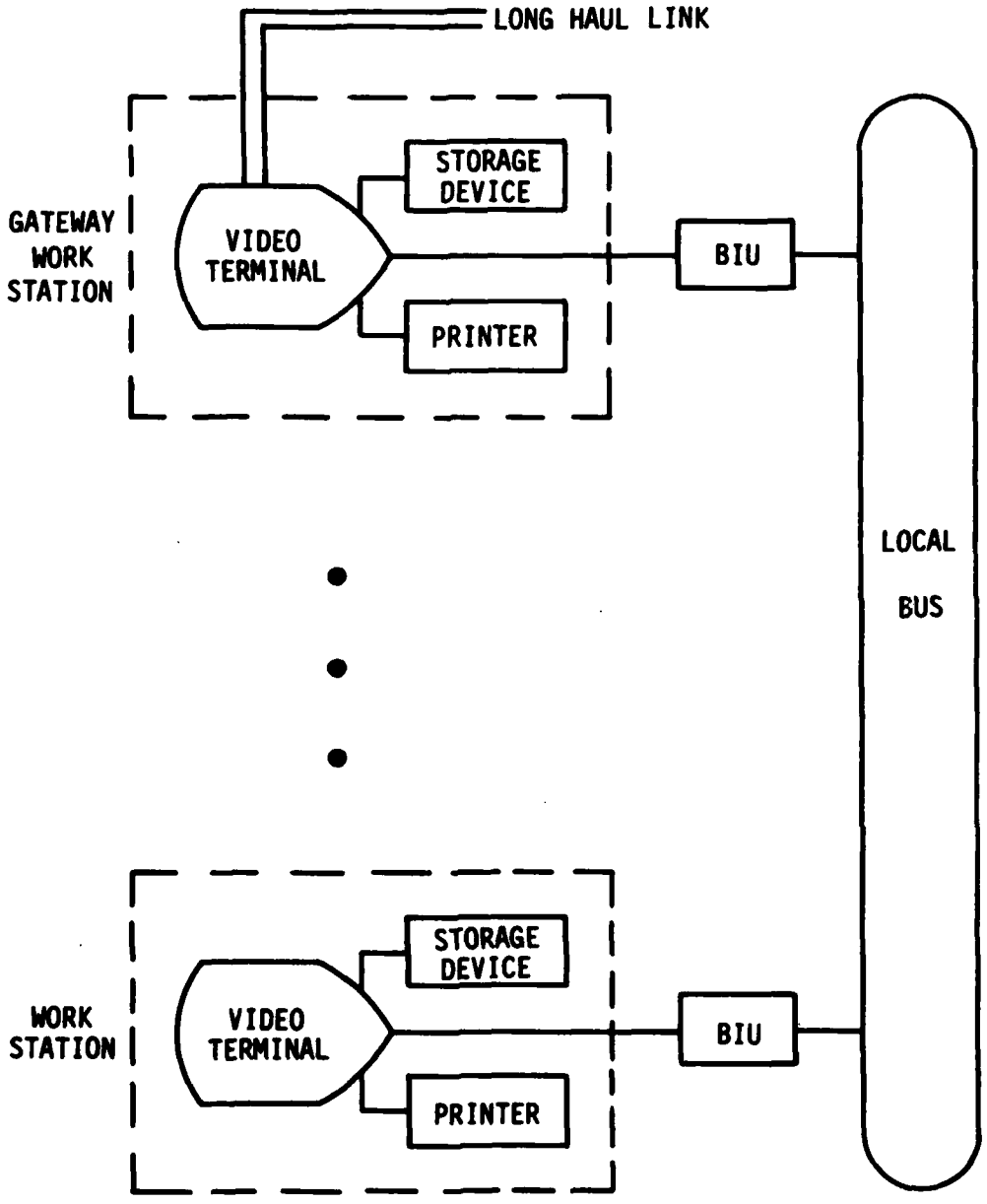
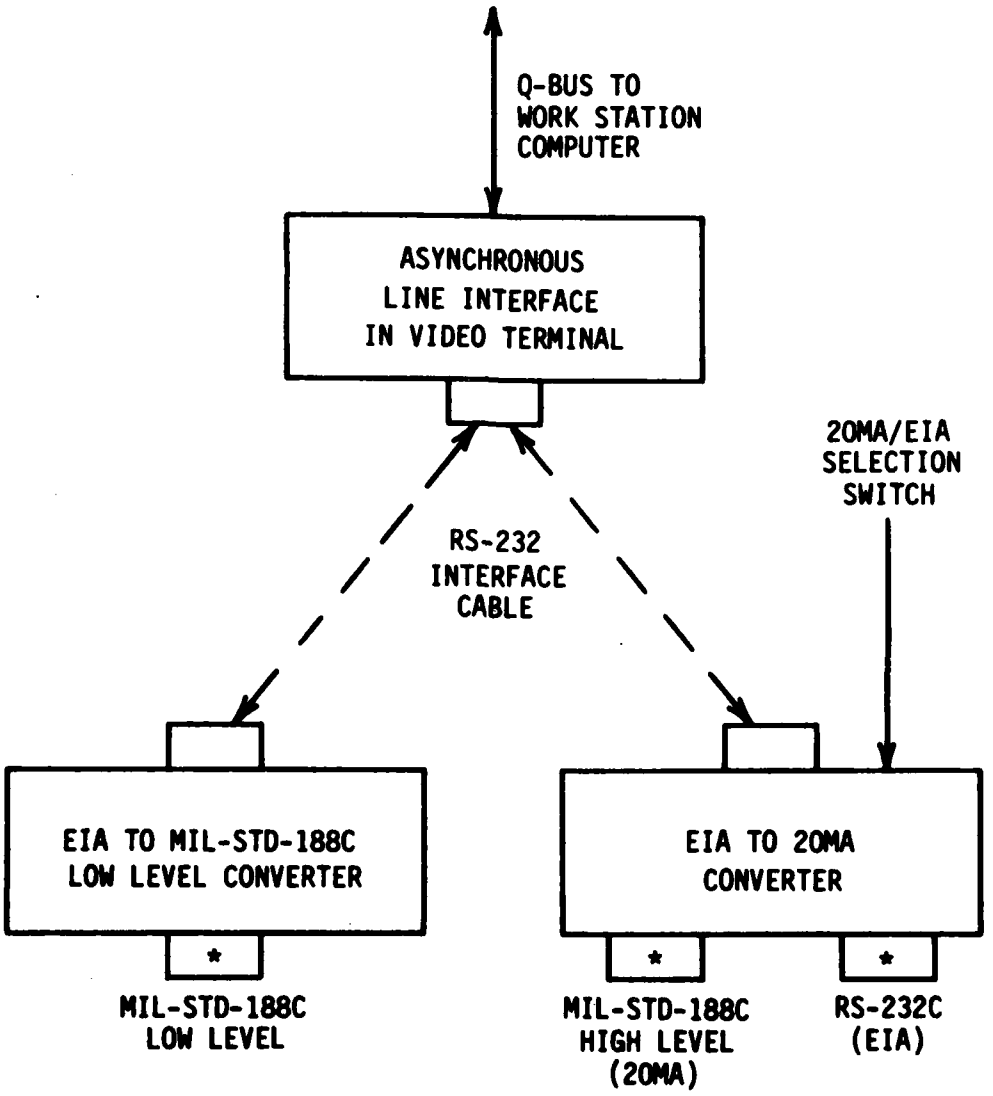


Figure 1. JAMPS OED System Diagram.



*LONG HAUL DATA LINK PORTS

Figure 2. Long Haul Interface

it is reformatted by an off-line batch program on the PDP-11/70, yielding another kind of tape, known in the JAMPS community as "the source tape" (figure 3). This "source tape" serves as the primary form of input to JAMPS off-line programs. There is concern that the continued existence of JAMPS may be jeopardized if the Army were to stop producing JINTACCS tapes or if the Langley Data Processing Center were to discontinue the maintenance of the program which reconstructs the JINTACCS data on "source tapes." The JAMPS project personnel would prefer to input the JINTACCS tape directly, bypassing the Data Processing Center at Langley AFB altogether.

Off-line (IBLD) programs in JAMPS read the "source tape" while generating object data base files for the disk contained in each work station. Other off-line JAMPS software checks the consistency and completeness of this disk-resident data. When it is determined that the disk-resident files are properly populated, the operator is permitted to load and enable the on-line programs used in real time operations. The JAMPS on-line software provides real time display and communication capabilities. The large disk-resident files which are generated off-line remain unchanged during on-line operations. Successful operation of the on-line functions depends on the existence of an error-free data base on disk. Error checking of the data in disk-resident files is not performed on-line because this would degrade response-time performance. One of the most significant features of JAMPS is that the off-line and on-line programs need not be modified as the result of changes to JINTACCS message types and formats.

During the past year, considerable effort has been expended to measure computer resource utilization by JAMPS on-line programs. As the result of these efforts, it has been determined that the responsiveness of the on-line software is being impaired because of excessive disk-access requests. To remedy this situation, MITRE personnel have proposed (along with other modifications) that the disk-file structures be reorganized in a more efficient manner.

1.4 ASSUMPTIONS

- Duplicating JAMPS software using Ada will be accomplished as an acquisition effort by a contractor.
- The Ada software for JAMPS will be procured in accordance with 300-series regulations. If the normal ESD procurement practices (800-series regulations) were followed instead, the software development cost estimates shown herein should be increased rather substantially, perhaps even doubled.

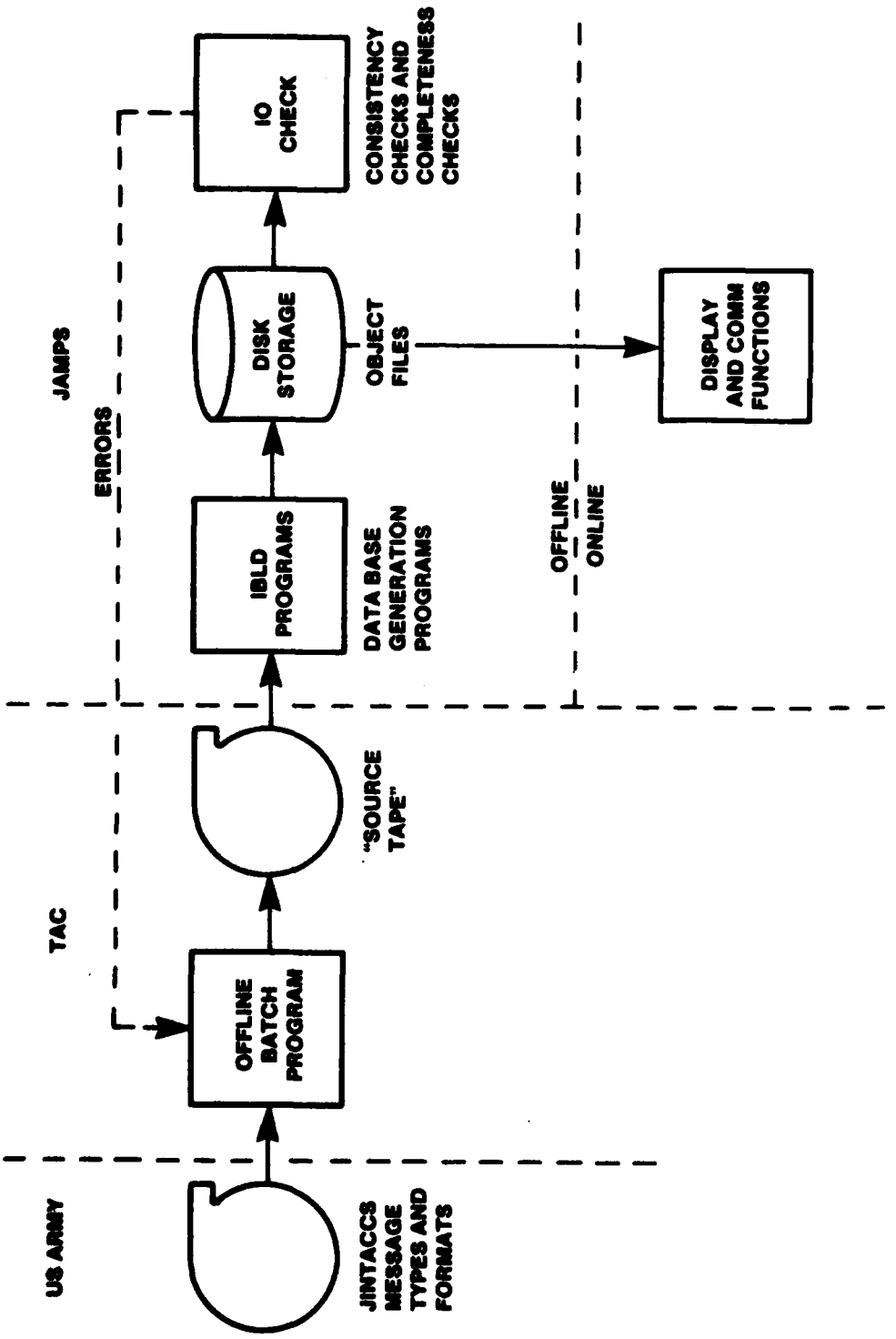


Figure 3. Data Base Initializaton Procedures for JAMPS

- The development of software in Ada will not be on a critical-path schedule for JAMPS because the existing software written in "C" will have already been transported to the MC68000.

- The requirements baseline used by the contractor will include the existing MITRE documentation for JAMPS, with amendments 1) to update the operator-interface description, 2) to include requirements for response-time performance and excess processing capabilities (e.g., for future growth), and 3) to include new requirements for a local area network.

- The contractor will receive considerable support from the Air Force in interpreting the system requirements.

- The existing allocation of functions between hardware and software in JAMPS will remain unchanged during the reimplementation using Ada. Therefore, only C level (not A or B level) documentation need be prepared by the contractor.

- The current hardware configuration, including the MC68000 computer, will be the basis for software development in Ada, and the main memory of the computer will be increased to 1-megabyte capacity.

- The off-line programs used by the Data Processing Center at Langley AFB to process the JINTACCS tape will be modified by the government to produce a new type of "source tape" format which substantially reduces the complexity of the off-line (IBLD) programs in JAMPS; it is assumed that existing JAMPS off-line functions which heavily utilize YACC and LEX functions will not be duplicated in Ada.

- Major emphasis will be given to software reusability during the reimplementation in Ada, and this will increase the contractor's costs for software development by 20%.

- The TCP/IP communications handlers available in the Berkeley UNIX 4.2 (estimated to be 6500 lines of "C" source code) can easily be incorporated into either the UNIX or the ROS operating systems used in the two alternative Ada Run Time Environments available today from Telesoft Corporation.

- The JAMPS software architecture will be fully redesigned by the contractor in order to take maximum advantage of desirable features inherently available in the Ada language.

- The number of source statements to be developed in Ada can be predicted accurately by applying conversion factors, based on the results from a recoding experiment, to extrapolate from the actual sizing data for the existing software written in "C".

- Programmer productivity in Ada will be moderately less than that with other higher order languages.

- As an early user of Ada, the JAMPS project is liable to attract widespread attention within ESD; therefore, the level of effort for contract monitoring support will be larger than normal for an undertaking of this size.

1.5 CONCLUSIONS

- The current status of Ada programming support environments for the MC68000, including but not limited to Ada compilers, is not adequate for duplicating JAMPS software in Ada today. However, the programming tools available from one software vendor, Telesoft, are expected to improve sufficiently over the next year so that the use of Ada for JAMPS can be undertaken in FY85. The Ada programming support tools currently offered by a second software vendor, Irvine Computer Sciences Corporation, are judged to be less adequate than those of Telesoft at this time, but will be of potential interest if redevelopment of JAMPS software in Ada begins late in FY85.

- The feasibility of using Ada to duplicate JAMPS software will be determined largely by the characteristics of the particular compiler implementations (and Ada run time environments) for the MC68000; the Ada language itself is not the problem. Certain optional features need to be added to the existing Ada run time environments in order for any of these environments to be amenable for use in the JAMPS application. The development schedules for such enhancements are not known precisely, but it is our assessment that the necessary improvements will have been completed by FY85.

- The total cost to duplicate JAMPS software in Ada is estimated to be $4.5M (1983 dollars). This estimate includes work to be performed by a software contractor, contract monitoring support, software redesign by the Data Processing Center at Langley AFB, consulting services, and

the acquisition of appropriate computing facilities and programming support tools for Ada software development. An upper bound for the total cost for reimplementation in Ada, based on various pessimistic assumptions, is $6.3M.

- The existing JAMPS software written in "C" will be awkward to modify for use in other systems because the record formats for disk files are not explicitly defined (in "C" source code) and are only partially described. In addition, the "C" software was not developed with reusability in mind, and therefore machine dependencies, compiler dependencies, and run time environment dependencies have not been carefully isolated and encapsulated in separate modules with appropriate annotations and documentation.

- The reliability of object code generated by incomplete versions of the compilers currently available for the MC68000 is fairly good.

- Early indications suggest that the response-time performance of Ada-compiled programs in the MC68000 will be satisfactory for use in JAMPS.

- A separate Ada software development facility will be needed for JAMPS; the memory size of the JAMPS MC68000 (and disk unit) is not adequate for an Ada compiler which must have access to extensive software libraries (source/object code).

- Software development in Ada on the MC68000 will be impeded by the unavailability of suitable tools for configuration control and performance measurement, and therefore the Air Force will be obliged to develop its own tools for these purposes.

- JAMPS offers an unusually good situation for early use of the Ada language because the system already exists and any reimplementation in Ada will not be driven by the tight development schedules which often characterize ESD procurement efforts. The amount of code to be developed in Ada is far less than the average size C3I system. The potential for reuse of JAMPS software in other systems is unusually high.

- The investment to date in "C" language software for JAMPS is very substantial (15 man-years of MITRE effort and 5 man-years of Air Force personnel time), perhaps 3 man-years of which would be recoverable if JAMPS software were to be reimplemented in Ada.

1.6 RECOMMENDATIONS

- Do not undertake an effort to duplicate JAMPS software in Ada until such time as a suitable compiler with an appropriate run time environment is fully available and has been validated by the Ada Joint Program Office. In addition, other necessary Ada programming support tools should also be readily obtainable.

1.7 SCOPE

The remainder of this report is organized into four major sections as follows:

- Feasibility of duplicating JAMPS software using Ada
- Estimated number of source statements of Ada to be developed for JAMPS
- Manpower, schedule, and cost estimates
- Advantages and disadvantages of reimplementing JAMPS software in Ada.

SECTION 2

FEASIBILITY OF DUPLICATING JAMPS SOFTWARE IN ADA

Issues surrounding the feasibility of duplicating JAMPS software in Ada are presented in the following order:

- Availability of Ada compilers for the MC68000
- Availability of Ada programming support tools for the MC68000
- Appropriateness of existing Ada run time environments
- Predicted run time performance
- Compiler reliability
- Physical limitations of Telesoft's Ada compilers

2.1 AVAILABILITY OF ADA COMPILERS FOR THE MC68000

Telesoft Corporation[2] and Irvine Computer Sciences Corporation (ICSC)[3] are the only firms which have formally announced the availability of Ada compilers for the MC68000. The Telesoft and ICSC Ada compilers do not presently support the full Ada language and therefore are not ready to be validated by the Ada Joint Program Office. Accordingly, these compilers are unsuitable for use in DoD acquisition programs (per draft DoD Directive 5000.31[4]). However, both Telesoft and ICSC claim that their compilers will be ready for validation in 1984.

As will be explained further on, Telesoft has many different versions of the Ada compiler for the MC68000 and not all of them appear to be suitable for the JAMPS application. The order in which the many different versions of the Telesoft compiler will reach completion and undergo validation tests is unclear. It is entirely possible that some of the Telesoft compiler versions will never be fully developed and therefore will never be validated. Finally, there is only one version of the ICSC Ada compiler for the MC68000 and its development schedule is also undetermined.

2.2 AVAILABILITY OF ADA PROGRAMMING SUPPORT TOOLS FOR MC68000

DoD has issued a nonbinding set of guidelines[6] for support tools to be used in conjunction with compilers for Ada software development. In table 1, these DoD guidelines are compared with the tool capabilities offered by Telesoft and by ICSC for the MC68000. This information in table 1 is based upon sales brochures and telephone communications with marketing representatives. It appears that the lack of tools for configuration control and performance analysis (static and dynamic analyzers) will significantly hinder any attempts to duplicate JAMPS software in Ada. Such tools could be procured directly from a software vendor, but this would significantly increase the budget required for recoding of JAMPS software in Ada.

2.3 APPROPRIATENESS OF EXISTING ADA RUN TIME ENVIRONMENTS

JAMPS software can easily be recoded in Ada. Nevertheless, recoding in Ada may not be a practical course of action. The appropriateness of the existing Ada run time environments, more than anything else, will determine the feasibility of duplicating JAMPS software using Ada. As a programming language, Ada has eliminated many of the fundamental design choices which traditionally have been made by applications software designers. Although the syntax of the Ada language has been rigorously standardized, many of the features of Ada run time environments have been left to the discretion of the individual compiler designer. For this reason, the selection of a compiler based on an inappropriate run time environment can kill a project before a single line of applications code has been written. The question "can we use Ada" is meaningless because there are many degrees of freedom in the design of Ada compilers. It is more reasonable to ask "can we use a specific compiler (i.e., one of the many different Ada Telesoft compilers or the ICSC compiler targeted to the MC68000) for the JAMPS project?"

The ensuing discussion of run time environments is organized as follows:

- Definition of "Ada run time environment"
- Formal requirements for Ada run time environments
- Optional features for Ada run time environments
- Comparison between JAMPS' needs versus capabilities available with existing Ada run time environments

Table 1

Survey of Existing Ada Programming Support Tools for MC68000

Software Tool	Telesoft/LabTek Status	ICSC Status
Text Editor (for entering and modifying Ada source code)	Available now from Telesoft	Available now
Pretty Printer (for printing text in legible formats)	Available now from LabTek with WICAT version of MC68000	Unavailable
Compiler (for translating Ada source code into object code for execution on MC68000)	Available in partial form now; should be fully available in 1984	Available now in partial form; should be fully available in 1984
Linkers (for resolving interfaces between separately compiled modules, forming executable programs)	Available now from Telesoft	Available now from ICSC
Loaders (for loading executable programs in both host and target computers)	Available now from Telesoft.	Available now
Symbolic debugger (for snapshots, traces, etc.)	Under development at Telesoft; due for delivery in 1984.	Not available; not in development.
Static analyzer (for data item set-use listings, cross-reference maps, calling relationships)	Not available; not currently in development.	Not available; not in development.
Dynamic analysis tools (frequency analyzer, timing analyzer)	Generally not available; LabTek offers a simple routine for measuring single-thread execution times.	Not available; not in development.

Table 1 (Concluded)

Survey of Existing Ada Programming Support Tools for MC68000

Software Tool	Telesoft/LabTek Status	ICSC Status
Terminal Interface Routines	Available now from Telesoft	Available now
Command Interpreter (accepts commands to invoke tools)	Available now from Telesoft	Available now
Configuration Management Tools	Not available; not currently in development	Not available; not in development
Stub Generator (to insert dummy program elements	Not available; not currently in development	Not available; not in development

2.3.1 Ada Run Time Environment Defined

The following material is quoted from "The Ada Run Time Environment," a lecture given by Dr. Joseph K. Cross, Sperry UNIVAC, at an AdaTEC Meeting, held in Dallas, Texas, on 20 October 1983:

"An Ada run time environment is, roughly, the set of target-machine facilities that an Ada compiler can use to carry out the run-time operations required by Ada programs. Those facilities consist of the instruction set provided by the physical target machine, possibly with additions and deletions. Additions to the facilities provided by the physical target machine's instruction set are generally provided by some predefined software, such as an executive, that in the compiler's eyes, might as well be implemented in hardware. Other additions to the physical target machine's facilities can be provided by additional hardware, such as an array processor, and by user microcode. Deletions from the physical target machine's facilities generally result from a conscious decision not to use some capability, generally in the interest of safety or simplicity. For example, after it had been decided to use a certain executive in the target machine, it might be determined that all code emitted by the Ada compiler will only run in the task state; then, the privileged instructions in the hardware's instruction set would not be usable by the Ada compiler, and would therefore not be part of the run time environment."

"After the target hardware has been chosen [i.e., the MC68000], after any predefined software [i.e., UNIX operating system, math routines, etc.] have been specified, and after all restrictions and conventions have been imposed, the compiler sees as a new target machine the virtual target machine for which all code is actually to be emitted. To the compiler, this virtual target machine is as different from the original physical target as if it were a different box: for example, the virtual box may have a SINE instruction while the physical machine did not, and the physical machine might let any register be used as a stack pointer while the virtual machine reserves register 15 for that purpose."

2.3.2 Formal Requirements for Ada Run Time Environments

The Ada Language Reference Manual[6] levies a minimal number of requirements on run time environments. These requirements are summarized as follows:

- Operations -- (e.g., addition, comparison, assignment, indexing) on various kinds of values (e.g., integer, floating and fixed point Boolean, record, array). Branches (GO TO, IF, CASE, LOOP, Subprogram CALL, and Return) are also required.

- o Tasking -- (creation/destruction of tasks, activation/abortion/termination of tasks, execution (start/stop) and rendezvous (simultaneous synchronization and data interchange)).

- o Input/Output -- (sequential, direct access, text, and low-level I/O).

- o Exception Processing -- (Ada requires that certain errors be detected at run-time, such as an attempt to assign the value 11 to a variable that has been declared to hold only values between 1 and 10. Such a run-time error is called an exception, and the result of an exception is to transfer control to a user-specified exception handler).

- o Memory Management -- (ability to store and retrieve values).

2.3.3 Optional Features for Ada Run Time Environments

This section provides examples of the facilities that a run time environment may provide over and above those that are required as described in section 2.3.2. The list is illustrative and does not attempt to exhaust the full set of possibilities.

- o Interrupt Handling. In Ada, an interrupt is treated like an entry call from an invisible task of very high priority. Hence a run time environment can satisfy the letter of the law by treating interrupts just like any other tasking operation. In some cases, JAMPS might need some form of expedited dispatching for such things as character-echo at the display console; interrupt handling tasks may be given control directly upon receipt of the interrupt, and without intervening enqueuing, scheduling decision, and dequeuing of a request. Also, advantage may be taken of the hardware register-saving capabilities.

- o Fancy Memory Management. Use of overlays, nonresident data, and garbage collection.

- o Distributed Processing and Multiprocessing. The virtual machine on which an Ada program runs may have more than one processor; the Ada language definition leaves these issues up to the compiler designer.

- o Multiprogramming. Various processing priorities may be assigned individually to Ada task programs and the run time environment may provide preemptive scheduling capabilities.

- o Fancy I/O. Asynchronous I/O, formatted I/O, use of key words.

2.3.4 Preliminary Selection of Run-Time Environments for JAMPS

At the outset, there are four versions of the Telesoft Ada compiler to be considered. These four versions are differentiated on the basis of their respective run time environments as follows:

- o P-code interpreter under the UNIX operating system
- o P-code interpreter under the ROS operating system
- o Machine code executed under UNIX (analogous to the ICSC compiler)
- o Machine code executed under ROS

Two versions of the Telesoft Ada compiler generate P-code (a Pascal derivative). The front-end of the compiler generates P-code on a host computer (VAX, IBM 370, or MC68000) and this intermediate form is then downloaded into the target computer (the MC68000 in JAMPS) where it is interpreted on-line by the "Run Time Kernel" (i.e., run time environment software). This means that the final part of translation and execution occur more or less simultaneously. For real-time applications, the use of an interpreter in an Ada run time environment would be a mistake; the response-time performance with an interpreter would be intolerably slow.

Two more recent versions of the Telesoft-Ada compiler plus the ICSC compiler all emit low-level (machine language) outputs. These compilers convert Ada source code into object programs in machine language form which can then be downloaded (from the host) into the target computer for execution under the control of an Ada run time environment. This second type of compiler is potentially of interest for use in development of software for JAMPS.

Telesoft provides two different run time environments based on the UNIX and the ROS operating systems, respectively. Programs compiled via the ICSC-Ada compiler will only execute under UNIX. Telesoft-UNIX requires a lot more memory (260 Kbytes) than ROS (80 Kbytes) in the target computer. UNIX is reputed to be significantly slower and more unwieldy for real time applications than ROS. Nevertheless, the use of UNIX in other systems is so widespread that in the interest of promoting widespread reusability of JAMPS software, the run time environment based on execution of machine code under UNIX should be seriously considered.

2.3.5 Comparison of JAMPS Requirements Versus the Characteristics of Selected Ada Run Time Environments

Table 2 shows a comparison between JAMPS requirements versus capabilities available with various Ada run time environments. It is assumed that the JAMPS program will have been compiled into machine language format. It is further assumed that all of the formal requirements for an Ada run time environment (see section 2.3.2) will have been satisfied by the time any Ada compiler is validated by the Ada Joint Program Office. Hence, only the optional features (such as those mentioned in section 2.3.3) need be carefully analyzed in this report. Unless stated otherwise in table 2, the characteristics of the ROS- and UNIX-based run time environments are believed to be the same.

Inspection of table 2 will reveal that even the most promising of the Ada run time environments (i.e., Telesoft's environment which controls the execution of machine code under ROS) does not come close to satisfying the needs of JAMPS at this time. However, since Telesoft claims that it plans to implement new features which will rectify most of the deficiencies noted herein, it will be reasonable to reevaluate the situation in six to nine months*. The current status of the ICSC run time environment suggests that it will not be ready for use by JAMPS programmers for quite some time. The information shown in table 2 is based on telephone conversations with representatives of Telesoft and ICSC, and cannot easily be substantiated because published reports describing the features of the various alternative run time environments are not available from the vendors.

*Telesoft, a relatively small software house, is believed to have a current backlog of 20 contracts for retargeting its Ada compiler to various different types of computers. It appears that Telesoft is not in a position to undertake an additional contract in the short-term to develop a customized run time environment for JAMPS.

Table 2

Comparison of JAMPS' Requirements Versus Capability of Existing Ada Run Time Environments

JAMPS Requirements	Telesoft Run Time Environment	ICSC Run Time Environment
Task Management Requirements	See below	No capabilities for tasking at this time. Future plans for task management are undefined.
Multiprogramming	Not available at this time; the Priority Pragma is supposed to be implemented with ROS during 1984.	
No restrictions on the number of different separately-scheduled tasks	Currently limited to 32 tasks; this restriction is is due to be removed in 1984.	
Multitasking	Supports sharing of data and pointers between tasks (at most, 256 words can be exchanged).	
Expedited dispatching (character echo)	Currently available, with 100 usec delay as an upper bound.	
No restrictions on size of packages, tasks, subprograms	Currently, individual packages are restricted to 32 Kbytes, but this limitation is expected to be removed during 1984.	
No restrictions on the number of "main" programs, with provisions for communications between "main programs."	"Main" programs cannot run concurrently; restricted to one 32-bit word for communication between two main programs.	

Table 2 (Continued)

Comparison of JAMPS' Requirements Versus Capability of Existing Ada Run Time Environments

JAMPS Requirements	Telesoft Run Time Environment	ICSC Run Time Environment
Memory Management Requirements:		
Detection of 80% saturation of dynamic memory areas	Not available. JAMPS software designers must provide a workaround in their applications software.	Not available
No restrictions on number of levels of nested procedures	Currently, dynamic memory is exhausted after four levels of nesting (when machine runs out of registers used for this purpose). The compiler is being redesigned to circumvent this limitation.	Unknown
No arbitrary limit on dynamic memory space allocated to individual task programs.	Currently fixed at 4 Kbytes/task, 32 Kbytes/package. In subsequent compiler releases, the programmer will be permitted to statically assign as much dynamic memory space as his program requires.	No restrictions on dynamic memory space.
File lock/unlock services	Can use Ada's Rendezvous construct for this purpose.	Same as for Telesoft.

Table 2 (Continued)

Comparison of JAMPS' Requirements Versus Capability of Existing Ada Run Time Environments

JAMPS Requirements	Telesoft Run Time Environment	ICSC Run Time Environment
No restrictions on memory space for Access Types as seen by individual packages, tasks, subprograms	Not currently available, but restriction is due to be removed during 1984.	Same as for Telesoft.
Disk-file access either by serial or by indexed-sequential addressing techniques.	Currently available.	Currently available.
Operations		
Data manipulation capabilities for two dimensional arrays.	Currently supported.	Unknown
Can perform operations on disk records without having to move them from I/O buffer areas before hand.	Currently supported.	Unknown
Desirable feature: Can perform operations on low-level I/O data (e.g., communications data) without having to move the data from the I/O buffer areas beforehand.	Not supported. Low-level I/O buffer space is limited to 8 Kbytes.	

Table 2 (Continued)

Comparison of JAMPS' Requirements Versus Capability of Existing Ada Run Time Environments

JAMPS Requirements	Telesoft Run Time Environment	ICSC Run Time Environment
I/O Management Requirements		
Asynchronous I/O for character and block-oriented devices.	Currently available.	Not currently available.
Text I/O	Currently available.	Currently available.
Low-level I/O	Currently available.	Currently available.
Device driver for a telephone modem connection.	Not currently available.	
Other device drivers (floppy disk, CRT, Winchester disk)	Currently available.	Unknown
Date/time Support	Currently available; time to nearest second.	Currently available.
Disk I/O driver accommodates individual records which are up to 8 Kbytes in length.	Currently available.	Unknown
System Initialization Requirements		
Flush out of I/O buffer areas (desirable feature but not a requirement)	Not available. Programs which rely upon use of uninitialized variables are erroneous.	Not available.

Table 2 (Concluded)

Comparison of JAMPS' Requirements Versus Capability of Existing Ada Run Time Environments

JAMPS Requirements	Telesoft Run Time Environment	ICSC Run Time Environment
Capability for assigning logical units to physical devices (consoles)	Unknown	Unknown
Optimization Support		
Can optimize run time system either for faster execution or for reduced memory use.	Currently available with ROS	Unknown
Ability to interface Ada-compiled programs with other programs written in "C" language; although not an absolute necessity, this feature is highly desirable.	Pragma Interface to to other languages is not supported; nor are there any immediate plans to implement this Pragma.	Currently available.
Ability to use in-line code in lieu of procedural calls to the run time environment.	In-line Pragma is not currently available but is supposed to be supported in 1984.	Unknown
Suppression of Range Checks in order to improve response time performance.	The Suppress Pragma is currently supported by Telesoft's compiler.	Currently available.

2.4 PREDICTED RUN TIME PERFORMANCE

The expected performance of Ada-compiled programs in the embedded MC68000 computer is a matter of paramount concern. Some of the early Ada compiler implementations are reputed to produce very inefficient object code. There are no formal requirements placed on Ada compilers for object code efficiency.

Run time performance will be discussed as follows:

- o Benchmark measurements to determine execution times
- o Context switching times
- o Memory use

2.4.1 Benchmark Measurements to Determine Execution Times

One very popular yardstick for comparing the performance of higher-level languages in various microcomputers is a simple benchmark program based on the Sieve of Eratosthenes[7]. This program finds all of the prime numbers between 3 and 16381.

The benchmark test results shown in table 3 are only one crude indication that Telesoft's Ada/ROS compiler implementation will satisfy the response time requirements for JAMPS. It is assumed that the existing software written in "C" language will perform satisfactorily in the MC68000 and that on the basis of the timing data shown in table 3, the C code and Ada/ROS programs can be expected to run roughly at the same speed. It is unknown just how much degradation in response time performance can be expected if ROS were to be replaced by UNIX.

2.4.2 Context Switching Times

The time required to respond to an interrupt, suspend the current task and schedule the execution of another task is referred to as "context switching time." In clumsy run time environments based on operating systems intended for use in software development systems (i.e., not for use in real-time embedded computer applications), context switching times are typically 5-10 milliseconds in length; for some types of real-time applications, context switching times of this length would be intolerable. Telesoft claims that its Ada/ROS run time environment for the MC68000 will accomplish context switching in 0.5-1.0 milliseconds, and this length of delay appears to be quite reasonable for the JAMPS application.

Table 3

Benchmark Test Results Using the Sieve of Eratosthenes[7]

Computer	Operating System	Language	Execution Time (seconds)
MC68000, 8 MHz (Sun pm 68K)	ROS	Ada (Telesoft)	4.4
MC68000, 8 MHz HP-9830	ROS	Ada (Telesoft)	5.0
MC68000, WICAT, 150 WS	MCS/UNIX	C (Johnson)	4.71
MC68000, Charles River 68	UNOS	C	6.3
MC68000, 8 MHz EXOP MACS	Not available	C (Whitesmiths)	9.82
MC68000, 8 MHz	Not available	Assembly	0.49

2.4.3 Memory Use

The on-line and off-line JAMPS software written in "C" amount to 9,953 and 4,944 source statements of C source code, respectively; (see table 5 in section 4.4). On the basis of a recoding experiment described in section 4.3, it appears that Ada requires 46% more source statements than "C" for equivalent functions. Hence, 14,531 Ada source statements will be required for the JAMPS on-line functions (a separate memory overlay). On the basis of this same experiment, on the average, each complete Ada source statement generated 24.8 bytes of object code. It follows that the online applications software will occupy 360 Kbytes* (14,531 x 24.8 = 360K) of memory. The UNIX and ROS run time environments require an additional 260 Kbytes and 80 Kbytes, respectively. Therefore, JAMPS software written in Ada will require a main memory size of .75 to 1.0 megabytes, and this does not appear to be a cause for concern.

2.5 COMPILER RELIABILITY

The object code generated by immature compilers frequently doesn't execute properly (i.e., is unreliable). However, Telesoft has already sold 350 of its Ada compilers, and their widespread early use has allowed these compilers to mature at an unusually rapid pace. The consenus is that Telesoft Ada compilers are reliable enough to be useful, although workarounds are frequently required because significant features of the Ada language are not yet supported. Singer-Librascope[8], for example, claims to have successfully used a Telesoft Ada compiler for the MC68000 on two different acquisition projects.

Error diagnostics generated by immature compilers are frequently meaningless to applications programmers. However, at the AdaTEC meeting in Dallas (October 1983), Telesoft successfully demonstrated that their compiler underlines the offending clauses in erroneous Ada source statements; references to applicable sections of the Ada Language Reference Manual were observed to be appended to the faulty source statements as well. Nevertheless, while using the Telesoft Ada compiler on the WICAT computer at MITRE, a number of abstruse (unhelpful) error diagnostics have been observed and several compiler crashes have occurred. The apparent discrepancies between the demonstration in Dallas versus the Telesoft compiler performance at MITRE, Bedford are attributed to recent enhancements which are not yet available in the compilers released to the public.

*The existing JAMPS on-line software written in "C", which does not include local area network functions included in the sizing estimate for Ada-compiled programs, occupies 234 Kbytes.

In part, the feasibility of duplicating JAMPS software in Ada may hinge on the availability of quick maintenance support for Ada software development tools. Telesoft is a small firm (15 employees as of 1 February 1983), and it is uncertain whether an organization of this size can provide adequate maintenance support to many users. LabTek is an even smaller company which sells systems including WICAT machines (based on MC68000) in combination with Telesoft's Ada software development tools. In the past, LabTek has generously provided a lot of free advice to MITRE personnel when they were experiencing problems using the Telesoft Ada compiler. In many instances, these were programmer problems, not compiler errors. It has been more difficult to obtain answers from Telesoft directly. However, Telesoft does seem to be distributing improved versions of its compilers at intervals of once every two to three months.

2.6 PHYSICAL LIMITATIONS OF THE TELESOFT COMPILER

The limitations of the Telesoft Ada compiler have not been fully determined. However, users of the Telesoft computer do not seem to be voicing many objections in this regard. The following data summarizes several telephone conversations with marketing representatives:

- Maximum number of lines of source code per compilation module greatly exceeds any requirement which JAMPS might impose, and therefore is not regarded as a potential area for concern.

- Maximum size of symbol table $\leq$ 128 Kbytes.

- Compilation speed < 300 lines of source code/minute.

- Maximum number of simultaneous compilations (in parallel on same host computer) = 1. This limitation is significant. The schedule for reimplementing JAMPS software in Ada needs to be stretched out because the number of programmers who can access the host computer will be quite limited. Alternatively, more than one host computer might be used, but that approach will compound the problems of software configuration control.

SECTION 3

COST FACTORS

Factors which need to be taken into account while preparing cost estimates for duplicating JAMPS software in Ada are discussed as follows:

- o System requirements specification
- o Contract monitoring support
- o Software reusability
- o Training
- o Software development facilities
- o Software architecture
- o Testing
- o Performance measurements
- o Redesign of programs which generate "source tapes"
- o Configuration management

3.1 SYSTEM REQUIREMENTS SPECIFICATION

The requirements specification for JAMPS was not prepared in accordance with military standards and in some ways is not entirely appropriate for use by an independent contractor for an acquisition program. However, with ample support from the government, an amended version of the requirements specification will suffice. The requirements baseline for JAMPS is expected to be fairly stable during the next several years.

Inasmuch as The MITRE Corporation has already developed a prototype model of the JAMPS system, it is anticipated that the amount of system engineering work to be accomplished by the contractor will be less than normal for a project of this size. Functional allocations between hardware and software have already been determined. The operator interface is well defined.

3.2 CONTRACT MONITORING SUPPORT

An unusually high level of contract monitoring support will be necessary for several reasons:

- It is assumed that an entirely new set of disk file structures will be defined for JAMPS, and this will not be an easy task. The new file structures should be designed to facilitate the use of Ada and must be considered when the government undertakes redesign of the off-line programs at the Langley Data Processing Center.

- Many things need to be done* to establish and monitor a contract involving the use of Ada; there is little precedent within ESD which can be drawn upon for these purposes.

- It is anticipated that the JAMPS efforts to use Ada will attract widespread attention within ESD. Documents describing the "software methodology while designing in Ada" and "lessons learned while applying Ada" will be required in addition to the standard forms of software documentation which are normally required of DoD contractors.

3.3 SOFTWARE REUSABILITY

The strictness of Ada compiler validation reduces, but does not eliminate, the problems of software portability and reusability. It costs more and takes longer to prepare software that is easily reusable, regardless of which programming language is used[9]. Programming standards and conventions which require isolation of machine dependencies (e.g., precision of fixed point arithmetic), compiler dependencies (e.g., differences in computational efficiency), and run time environment dependencies (e.g., expedited dispatching) must be rigidly enforced. It is intuitively estimated that designing for software reusability will increase development costs by 20% and that this will more than pay for itself if JAMPS software is to be reused in numerous other systems. There is no empirical data available to support this intuitive estimate.

*Preparation of RFP, IFPP, Proposal Evaluation Criteria, policy decisions relative to the use of Ada-based Program Design Language in formal software documentation, guidelines for Ada programming style (standards and conventions), etc.

It is noted that the existing software written in "C" was not prepared with reusability in mind, and, as a consequence, is expected to be fairly awkward to incorporate in other systems. It is also observed that the reusability of JAMPS software written in Ada will be diminished by any dependence upon the availability of specific features in an Ada run time environment which are not required by MIL-STD-1815A (Ada Language Reference Manual).

3.4 TRAINING

Program management and contractor personnel will probably need extensive training, both in the Ada syntax and especially in the software engineering principles upon which the Ada language is based. Every individual involved should have at least four weeks of formal training in Ada. Additional training in the preparation of reusable software is highly recommended. It is also suggested that the consulting services of an Ada expert be made available to the JAMPS project, especially during the early stages of development; this individual will critique the first attempts by JAMPS programmers to use Ada.

DoD policy disallows programmer training as an expense which can be directly charged to an acquisition project*. Nevertheless, during the transition period in which most programmers will be unfamiliar with the Ada language, it is inevitable that some amount of training will be required.

3.5 SOFTWARE DEVELOPMENT FACILITIES

The Telesoft Ada compiler requires a 1-megabyte main memory and a 15-megabyte disk storage unit. Configuration management of Ada source and object code files is expected to increase disk storage requirements even further. It is assumed that at least two standalone software development facilities will be acquired and maintained throughout the effort to duplicate JAMPS software in Ada. Much of the testing of Ada programs will take place on the host computer under the control of a symbolic debugger. Highly optimized Ada object code is generally undecipherable and cannot easily be patched in machine language format in the target computer. Therefore, the host computer (software development facility) should include several consoles to support parallel efforts by various programmers. Even

*The Director of the Ada Joint Program Office made this statement at the AdaTEC meeting in Dallas on 19 October 1983.

though only one programmer will be able to use the Telesoft Ada compiler at any one time, other programmers can do other chores, such as text editing, on the host computer (in parallel) while a compilation is underway.

3.6 SOFTWARE ARCHITECTURE

To take full advantage of the features inherently available with the Ada language, the software architecture for JAMPS must be fully redesigned. There is little point in recoding on a module-per-module basis with existing software written in "C"; this would result in Ada code that would be both inefficient and difficult to reuse in other systems.

On the basis of performance measurements, it has been determined that the response time performance of JAMPS is largely determined by the number of disk accesses required for various types of messages. To reduce the number of disk accesses, the existing disk-file structure needs to be changed extensively, combining certain files together and eliminating data stored redundantly in two or more files. The design of variant record formats should take into consideration the needs of generic input/output routines written in Ada. The record formats which currently exist in JAMPS will be difficult to accommodate in Ada.

3.7 TESTING

It is proposed that redesign of disk file structures will be accomplished in "C" prior to the reimplementation in Ada, and that the disk file structures used by programs written in "C" and in Ada will be kept exactly the same. This will reduce the risks associated with Ada by making it possible to partition the integration and testing of Ada software into two separate efforts:

- New (unproven) off-line programs written in Ada can be used to both populate and validate the disk-resident files; afterwards, proven on-line programs written in "C" can be used to verify that the disk files have been initialized appropriately.

- Proven off-line programs written in "C" can be used to populate and validate the disk-resident files; afterwards, new (unproven) on-line programs written in Ada can be exercised with high assurance that the data obtained from disk during on-line operations is correct.

This approach will permit integration testing in parallel for off-line and on-line programs written in Ada, and will reduce the overall time to accomplish the duplication effort. If this approach is not adopted, the software development schedules should be extended at least four months.

3.8 PERFORMANCE MEASUREMENTS

Special software tools should be developed by the Air Force for performance measurements on Ada programs in the MC68000 because such tools are not expected to be made available by software vendors in the immediate future. As a prerequisite for developing these tools, the government will need to acquire the source listings for the Ada run time environment which has been selected.

3.9 REDESIGN OF PROGRAMS WHICH GENERATE "SOURCE TAPES"

It is assumed that the government will undertake the redesign of off-line programs which process the "JINTACCS tapes." These revised programs will obviate any need for YACC and LEX capabilities as presently made available by UNIX to the IBLD programs in JAMPS. It is further assumed that the feasibility of using YACC and LEX in conjunction with Ada-compiled programs is extremely doubtful. Therefore, it would be necessary to duplicate YACC and LEX functions in Ada if the government is unwilling to assume responsibility for the proposed changes.

If JAMPS were redesigned to bypass any need for support from the Data Processing Center at Langley AFB, accepting the JINTACCS tape as input instead of the "source tape," then the functions currently provided by 2500 lines of "C" source code at the Langley Data Processing Center must somehow be accommodated in JAMPS. In addition, the YACC and LEX functions (2,670 and 2,820 lines of "C" respectively) and the IBLD functions (in JAMPS) which would otherwise have been absorbed by the Langley Data Processing Center (3,000 lines of "C") must be considered while scoping the size of the effort to duplicate JAMPS software in Ada.

3.10 CONFIGURATION MANAGEMENT

Telesoft does not currently provide tools for Ada software configuration control, nor does it intend to begin development of such tools during 1984. Nevertheless, it is highly desirable to have such tools for keeping track of dependencies between different versions of separately-compiled modules. Accordingly, the Air Force

will need to develop its own configuration management tools in support of JAMPS software development in Ada.

SECTION 4

ESTIMATES OF THE NUMBER OF ADA SOURCE STATEMENTS TO BE DEVELOPED

Estimates of the number of Ada source statements to be developed were determined in the following manner:

- The actual number of lines of "C" source code in the existing JAMPS system was computed with comment statements excluded.
- Adjustments were made to convert lines of "C" code into equivalent complete source statements written in the "C" language.
- Adjustments were introduced to reflect the expected effects of simplifying the functional requirements for JAMPS off-line programs via "source tape" redesign. Allowances were made for new local area network requirements not yet incorporated into the existing JAMPS software written in "C".
- A representative example of "C" code from JAMPS was recoded in Ada, yielding a planning factor for expressing/estimating Ada source statements in terms of equivalent (existing) "C" statements.
- Estimates for Ada source statements to be developed were computed by multiplying a conversion factor (from the recoding experiment) times the adjusted number of "C" source statements.

The validity of estimates which are based on an extrapolation from the results of a recoding experiment is highly questionable. It is quite possible that if the recoding experiment had been conducted by some other programmer and/or the representative example selected from existing JAMPS software to be recoded in Ada had been different, then the estimates for Ada source statements to be developed might change substantially. However, the more traditional approach for estimating the size of a software job, based on the cumulative sum of intuitive judgments about the sizes of various modules, is also prone to large errors. Regardless of which method is followed, an error of 25% while estimating "source statements" to be developed is entirely possible.

4.1 SIZING ANALYSIS FOR EXISTING JAMPS SOFTWARE

The results of a survey of JAMPS code written in "C" language are depicted in table 4; comment statements are not included. The data in table 4 corresponds with the "C" implementation on the DEC 11/23 and will be subject to minor changes while shifting over to the MC68000.

4.2 ADJUSTMENTS

The actual "source data" shown in table 4 has been adjusted in several ways, yielding the results depicted in table 5. The line counts in table 4 for "C" code include certain lines which contain nothing more than a single bracket, "[". For purposes of cost estimation, these lines should be treated just like "comment" lines, (i.e., ignored). On the basis of an examination of a representative example of the "C" code, it is estimated that 14% of all "C" source code lines shown in table 4 correspond to simple "brackets" and should be discounted accordingly.

The data shown in table 4 is expressed in terms of "lines of code." It is observed that in numerous instances, a single JAMPS source statement expressed in "C" occupies two or sometimes three lines. This is a matter of programming style. Some programmers prefer to write one source statement per line; others write individual source statements using several lines. For purposes of software cost estimation, it is more meaningful to use complete source statements, not just "lines of code." For this reason, the data in table 4 has been further adjusted to remove the efforts of multiple lines per source statement. On the basis of an inspection of JAMPS code, it is estimated that 16% of the lines of "C" should be discounted as "spillovers" from preceding lines. In summary, the "number of lines" of "C" code has been reduced by 30% (14% + 16%), while reexpressing the sizing data in terms of "source statements" (table 5).

It is estimated that 3000 lines of IBLD (off-line) "C" code can be eliminated by redesigning the "source tape" format. In essence, more of the burden of initializing JAMPS will be assumed by the Langley AFB Data Processing Center in the future. In the event that the government does not agree with this assumption, then the 3000 lines of IBLD code (i.e., off-line JAMPS functions) plus an additional 5540 lines of "C" code in UNIX (YACC, LEX) must be considered as part of the effort to duplicate JAMPS software in Ada. If JAMPS bypasses the Langley AFB Data Processing Center altogether, accepting the JINTACCS tape directly as input, then the functions

Table 4

Sizing Analysis for Existing Software Written in "C"

	Code Source Lines	Memory Utilization*
JAMPS Off-Line Programs		
IBLD Programs	7,736	Not available
Test Programs (consistency/ completeness)	2,327	Not available
	10,063	
JAMPS On-Line Programs		
Display	8,447	89,704
Communications	1,077	23,256
Remote Functions (repeated as in-line code)	3,694	121,536
	13,218	234,196 Bytes of on-line executable code, exclusive of UNIX.
Total for JAMPS	23,281	
Off-Line programs which generate "source tapes"	2,500	

*Memory utilization is based on "C" compiler for the DEC 11/23 computer.

Table 5

Sizing Data with Adjustments

	"C" Source Lines		"C" Source Statements
JAMPS On-Line Functions			
Existing Functions	13,218		
Allowance for Local Area Network Functions	1,000		
Equivalent lines to be recoded in Ada	14,218		9,953
JAMPS Off-Line Functions			
Existing Functions	10,063		
Less simplifications due to new "source tape" format	- 3,000		
	7,063	"Best Case"	4,944
If Langley AFB Data Processing Center is unwilling to revise the "source tape" format:			
Existing Off-Line JAMPS Functions	10,063		
LEX	2,870		
YACC	2,670		
Total off-line code to be duplicated in Ada	15,603	"In Between Case"	10,922
If JAMPS Bypasses Langley AFB Data Processing Center Altogether:			
Existing Off-Line Functions	10,063		
LEX	2,870		
YACC	2,670		
Programs which generate "source tapes"	2,500		
Total off-line code to be duplicated in Ada	18,103	"Worst Case"	12,672

currently performed by 2500 lines of "C" code at Langley AFB must be accommodated in JAMPS.

4.3 A REPRESENTATIVE EXAMPLE OF "C" RECODED IN "ADA"

A representative example of JAMPS software written in "C" has been recoded in Ada. The complete example in both languages is provided in Appendix A, and the results are summarized in table 6. Owing to the incompleteness of the Telesoft Ada compiler for the MC68000, many features of the language had to be avoided during compilation. Hence, the example had to be recoded using two different methods:

- Recoding without any restrictions (i.e., using features of the full Ada language). This method produced a conversion factor which can be used for determining the number of Ada-source statements to be developed on the basis of equivalent source statements written in "C".

- Recoding in Ada, circumventing deficiencies in the compiler whenever necessary. (Approximately one-third of the code written using the full Ada language produced error diagnostics during compilation; therefore, the code that will not compile appears as comments in the listing shown in Appendix A.) This second method produced code which could be compiled error-free, and the results can be used for estimating memory requirements for programs written in Ada.

The results of the experiment suggest that Ada as a language is more verbose than "C" because 1.46 source statements in Ada were found to be functionally equivalent to 1.0 source statements in "C". This conclusion is somewhat misleading because the number of executable source statements written in "C" and Ada were roughly comparable (134 versus 125, respectively). The greatest difference between the number of source statements using the two languages is attributable to the explicit data declaration statements which Ada, unlike "C", requires for defining the disk-resident data records.

In "C", data record formats are implicitly defined in the executable code rather than in explicit data declaration statements. During the recoding experiment, trying to fathom implicit record formats proved to be fairly difficult, leading to the conclusion that Ada software would be easier to maintain than "C" software.

Table 6

Conversion from "C" to Ada

	Source Statements	
	"C"	Ada
JAMPS Recoding Experiment	167	244
Sieve of Eratosthenes	23	34

Conclusion: 1 Source Statement of "C" source code = 1.46 source statements in Ada.

$$\frac{244}{167} = 1.46$$

Each source statement in Ada generates, on average, 24.8 bytes of object code (see appendix A).

4.4 ESTIMATES OF THE NUMBER OF ADA SOURCE STATEMENTS TO BE DEVELOPED

The estimated number of Ada source statements to be developed is computed in table 7. Depending on what assumptions are made about the "source tape" format, the estimated number of Ada source statements ranges from 23,939 to 35,222.

Table 7

Estimates OF Source Statements of JAMPS Source Code to be Developed in Ada

	"C"	Ada
Off-Line Functions		
Best Case	4,944	7,218
In-Between Case	10,922	15,946
Worst Case	12,672	18,501
On-Line Functions	9,953	14,531
Configuration Management Tools	1,000	1,460
Performance Measurement Tools	500	730
Totals		
Best Case	16,397	23,939
In-Between Case	22,375	32,667
Worst Case	24,125	35,222

Conversion Factor: 1 source statement of "C" = 1.46 source statements of Ada.

SECTION 5

ADA IMPLEMENTATION PLAN

A tentative plan for duplicating JAMPS software in Ada is presented below. Basic assumptions have been made as follows: (1) a contractor will undertake the bulk of the effort involved; and (2) 300-series procurement regulations and practices will be followed. If, instead, the standard ESD acquisition practices were adopted, based on 800-series regulations, it is expected that the cost of duplicating JAMPS software in Ada would rise substantially, perhaps even double.

5.1 MANPOWER REQUIREMENTS

Estimated manpower requirements for various tasks are shown in table 8. This table reflects recent experience[10,11] in applying the Ada language; the amount of effort expended during preliminary design is above average and the amount of effort consumed during integration and testing is below average when Ada is compared with other higher order languages (see figure 4). Inasmuch as JAMPS has already been written in "C", the contractor is not expected to undertake the usual level of effort for requirements analysis; such things as operator-interface studies will already have been completed by MITRE and need not be repeated. Figure 4 illustrates the differences between manpower allocations for a "typical" acquisition program and those assumed for the JAMPS implementation in Ada.

Very little is available in the way of data and models to support software conversion estimation[12]. Typical productivity during new development using higher order languages is 300 delivered source statements per man-month. If JAMPS software is completely redesigned and fully rewritten in Ada, then this would be tantamount to a new development effort -- not a software conversion job. Owing to the complexity of the language and the lack of programmers experienced in Ada, it will be assumed for purposes of this study, that programmer productivity in Ada will be 250 delivered source statements per man month.

Table 8

Ada Implementation Plan

Task	Description		Responsibility	Level* of Effort (man months)
1	Update JAMPS System Performance Requirements		Government	1
2	Update JAMPS User's Guide		Government	2
3	Prepare IFPP		Government	8
4	Prepare RFP/SOW		Government	8
5	Reevaluate Feasibility of Duplicating JAMPS Software in Ada (to be accomplished 7/1/84)		Government	1
6	Source Selection Support		Government	4
7	Contract Monitoring Support		Government	21
8	Redesign the Disk File Structures		Government	12
9	Redesign Existing Programs in "C" to Use New File Structures		Government	36
10	Redesign the Format of "Source Tape"		Government	3
11	Redesign Off-line Programs Which Generate "Source Tapes" (50% Redesign)		Government	9
12	Acquire, Install, Demonstrate, and Maintain Ada Software Development Facility		Contractor	12
13	Conduct Benchmark Measurements to Determine Efficiency of Ada Compiled Code		Contractor	4
14	Requirements Analysis		Contractor	7
15	Preliminary Design	On-line Programs for JAMPS	Contractor	26
16	Detailed Design		Contractor	18
17	Code and Debug		Contractor	15
18	Development Testing		Contractor	14
19	Validation Testing and Demonstration		Contractor	12

*Based on "optimistic assumptions"

Table 8 (concluded)

Task	Description		Responsibility	Level of Effort (man months)
20	Requirements Analysis	Off-line Programs for JAMPS	Contractor	4
21	Preliminary Design		Contractor	13
22	Detailed Design		Contractor	9
28	Code and Debug		Contractor	7
24	Development Testing		Contractor	7
25	Validation Testing and Demonstration		Contractor	6
26	Prepare Formal Validation Test Procedures		Government	6
27	Conduct Validation Tests		Government	4
28	Conduct (2) Timing/Sizing Analyses		Contractor	**
29	Document Potential Pitfalls to be Avoided While Attempting to Reuse JAMPS Software		Contractor	**
30	Document Ada Software Methodology Adopted		Contractor	**
31	Design, Code, Test Document Configuration Management Tools		Contractor	10
32	Design, Code, Test, Document Performance Measurement Tools		Contractor	4
33	Training in Use of Ada		Contractor	**
34	Training in Use of Ada		Government	0
35	Document Lessons Learned While Maintaining an Acquisition Program Using Ada		Government	3
36	Support to OED Demonstration		Government	3
37	Management		Contractor	21
38	Software Quality Assurance		Contractor	**
39	C5 Software Documentation		Contractor	**

**Included indirectly in man-month estimates for software development (tasks 14-25).

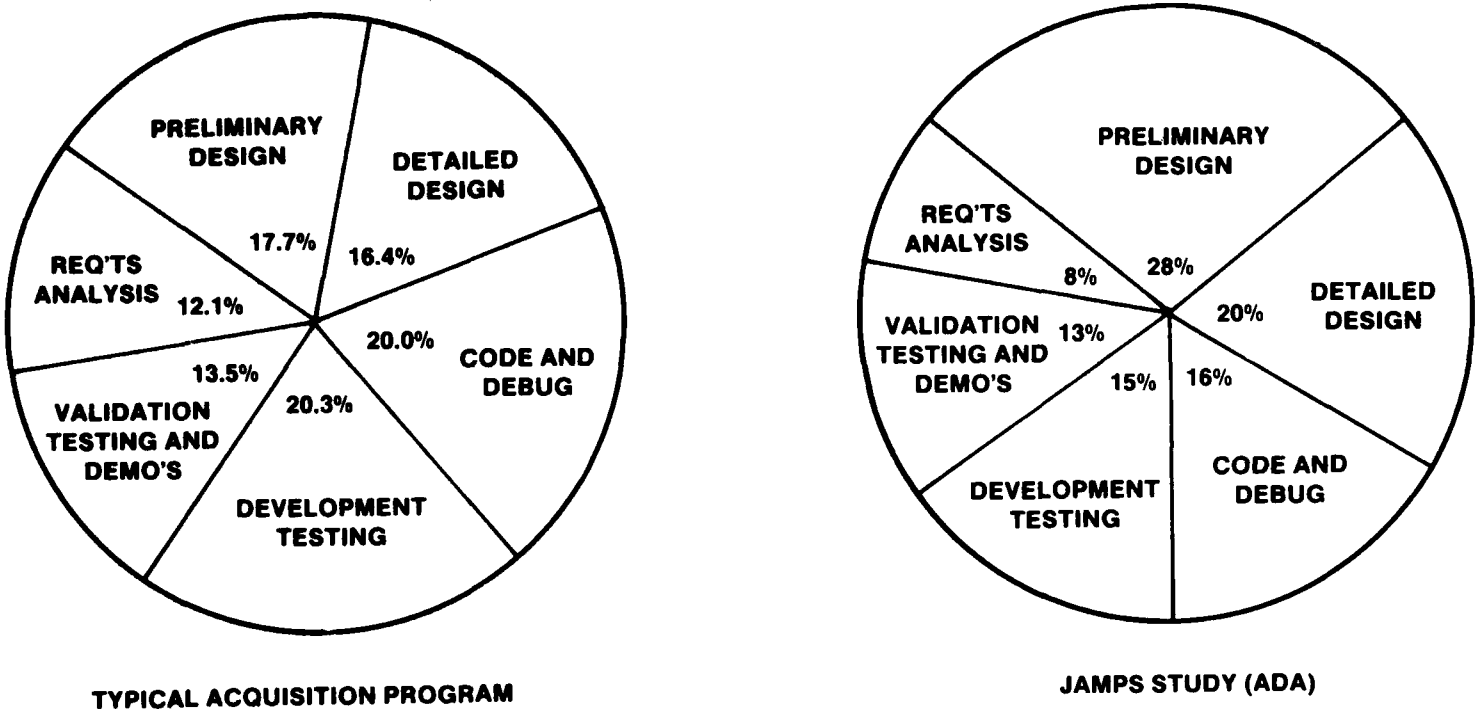


Figure 4. Compares the Work Breakdown for a Typical Medium Sized Acquisition Program (Based on a HOL Other Than Ada) with the Work Breakdown Structure Assumed for Duplicating JAMPS Software Using Ada. [12]

From section 4, it is estimated that a total of 23,939 Ada source statements need to be developed by the contractor:

On-Line Programs (redesign)	14,531
Off-Line Programs (best case)	7,218
Tools (new design)	2,190
	23,939

$$\frac{\text{23,939 source statements}}{\text{250 source statements per man-month}} = \text{120 mm for software development during the preliminary design through integration testing phases.}$$

From figure 4, preliminary design through integration testing constitutes 79% of the total software development effort; activities such as requirements analysis and formal demonstrations represent the remainder of the effort. Thus,

$$\frac{120 \text{ mm}}{.79} = \text{151 mm will be required for the total software effort by the contractor.}$$

The individual efforts by the contractor for development of on-line, off-line, and tool software are computed to be 92 mm, 45 mm, and 14 mm, respectively.

The 151 mm estimate for software development efforts by the contractor includes, among other things, tasks such as software documentation, timing/sizing studies, and software quality assurance. It does not, however, include the maintenance of software development facilities, contract administration, etc., and when these other factors (see table 8) are taken into account, the total manpower requirements for the contractor for all activities amount to 188 mm. Additional efforts to be performed by the government personnel are estimated to require 121 mm.

5.2 SCHEDULING REQUIREMENTS

A Gannt chart for duplicating JAMPS software in Ada is presented in figure 5. The tasks in the Gannt chart are defined in table 8. This chart indicates that the contractor's efforts will be completed in 21 months:

Requirements Analysis	months 0-4
Software design, development, and integration testing	months 4-19*
Acceptance testing, by the government	months 19-21
Operational effectiveness demonstration	months 21-23

It is assumed that off-line and on-line programs will be developed and tested independently of one another (see section 4.8).

*The results of the Cocomo model projections suggest that software design, development, and integration testing will only require 12.9 months (see section 6.2).

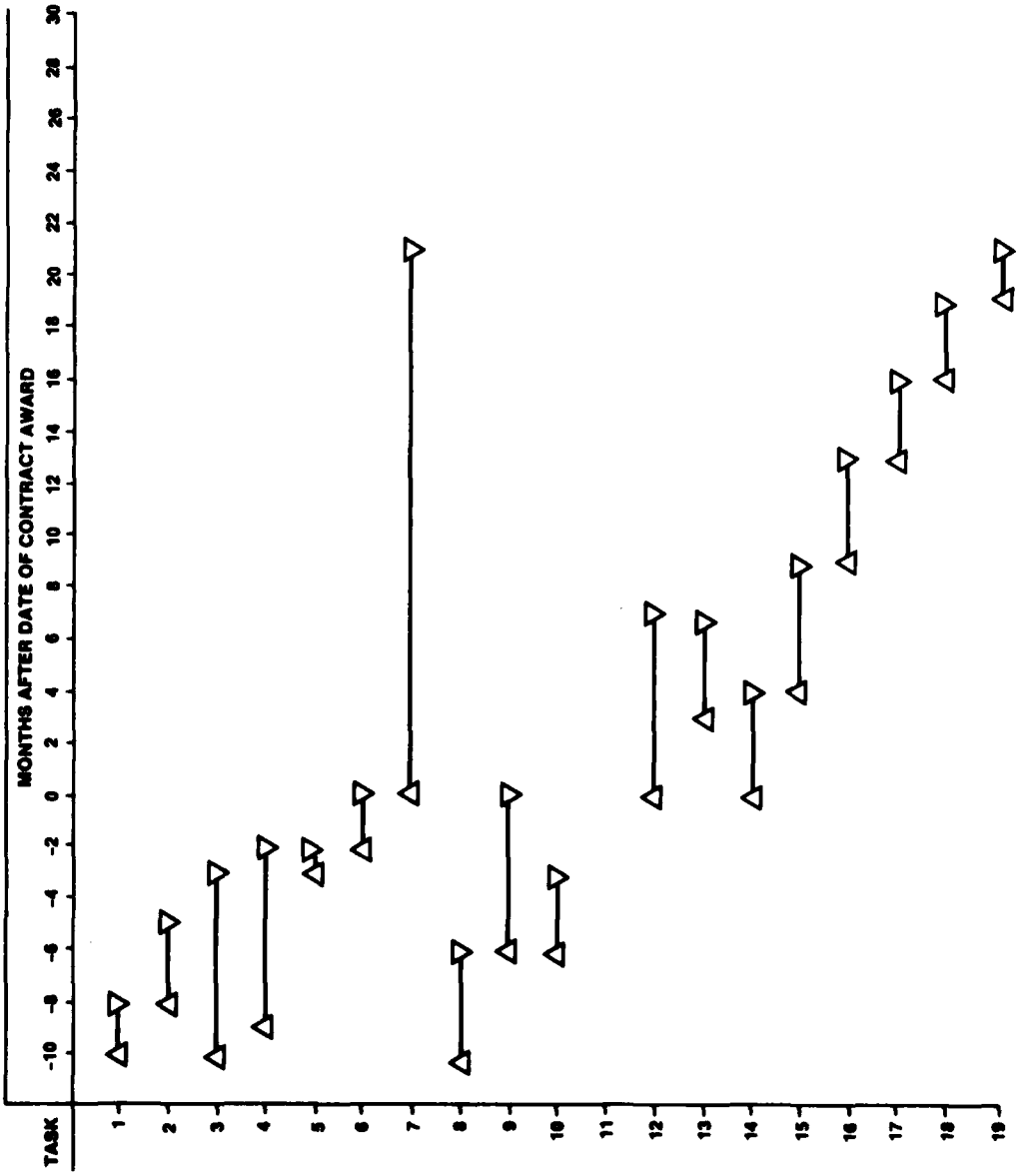


Figure 5. GANNT Chart for Duplicating JAMPS Software in Ada.

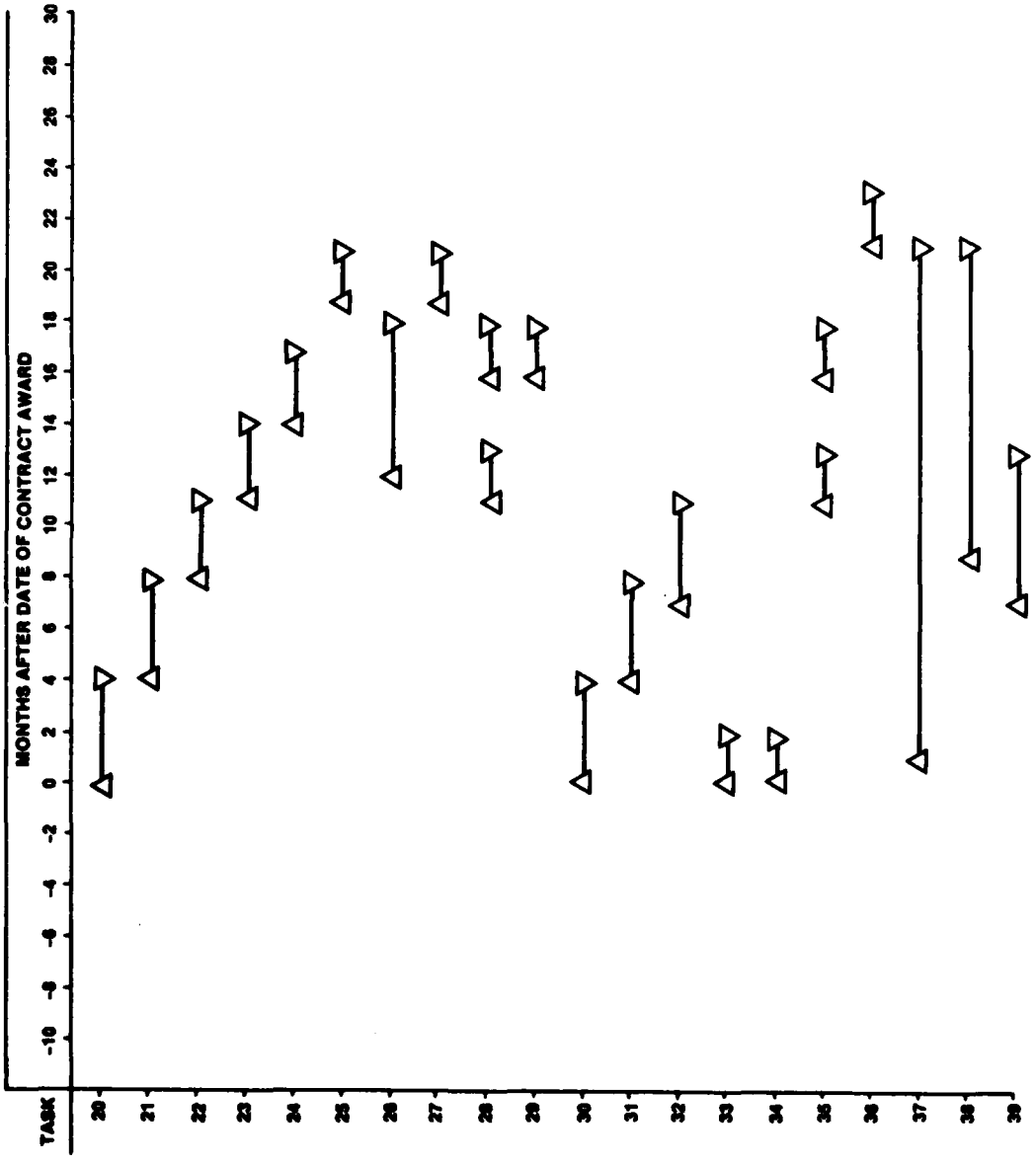


Figure 5. GANNT Chart for Duplicating JAMPS Software in Ada (concluded)

SECTION 6

COST ESTIMATION

The cost estimates for the contractor's effort to duplicate JAMPS software in Ada have been determined by two independent methods:

Method 1 - Software cost estimates have been computed by multiplying manpower estimates (see table 8) times assumed labor rates (which include general and administrative costs).

Method 2 - Prediction of software development costs via the Cocomo model. The second set of results will be used for purposes of comparison (i.e., for assessing the reasonableness of the estimates generated by Method 1).

6.1 METHOD 1: MANPOWER REQUIREMENTS MULTIPLIED BY ASSUMED LABOR RATES

According to Reference 13, microcomputer programmers were paid annual salaries during 1983 as follows:

Experience (Years)	Salary ($K)
0-2	24.0
2-4	27.7
4+	34.5

For purposes of this study, an annual salary of $30K will be assumed. To this figure, 128% needs to be added for overhead and 20% for general and administration expenses. Hence, the cost per man-month is computed to be $6,200.00

Optimistic and pessimistic estimates for duplicating JAMPS software in Ada are depicted in tables 9 and 10, respectively. Taking the average of the optimistic and pessimistic estimates, the total cost to duplicate JAMPS software in Ada is projected to be $4,486K.

Table 9

Optimistic Cost Estimate

	Man-Months	x $/MM	Initial Estimate via Method 1	Estimate with 20% Adjustment Because of Software Reusability Requirements and 6% Incentive Fee
Contractor	188	x 6,200	= $936K	$1,179K
Government	85*	x 9,166	=	$ 779K
Software Development Facility (Labtek[14])				$ 50K
Training by Consulting Firm				0
Consulting Services				$ 100K
Source Listing for Ada Run Time Environment				$ 13K
Management Reserve (25%)**				$ 530K
				$2,651K

Cost per source statement $\frac{\$2,651K}{23,939}$ statements = $110.74/source statement

Programmer Productivity = 250 source statements/mm.

*Task 9, Redesign of Existing Programs in "C" to use new file structures, has been omitted from the "Optimistic Cost Estimate".

**A management reserve of 25% is consistent with ESD practice for software redesign.

Table 10

Pessimistic Cost Estimate

	Man-Months	x $/MM	Initial Estimate via Method 1	Estimate with 50% Adjustment for Software Reusability and 6% Adjustment for Contractor Award Fee
Contractor	381*	x 6,200	= $2,362K	x 1.56 = $3,685K
Government	121	x 9,166		$1,109K
Software Development Facility				$ 50K
Training by Consulting Firm				$ 50K
Consulting Services				$ 150K
Source Listing for Ada Run Time Environment				$ 13K
Management Reserve (25%)				$1,264K
				$6,321K

Cost Per Source Statement = $\frac{\$6,321K}{35,222 \text{ Statements}}$ = $179.46

Programmer Productivity = 129.4 Source Statements/mm (Per Cocomo Model)

$\frac{35,222 \text{ Source Statements}}{129.4 \text{ s.s/mm}}$ = 272 mm for preliminary design through integration and testing

$\frac{272}{.79}$ = 344 mm total S/W development effort
+37 mm for contract administration etc.
381 total for contractor

6.2 METHOD 2: COST ESTIMATION VIA THE COCOMO MODEL

The Cocomo model[15] has been applied as a means of verifying the validity of cost and schedule estimates shown previously herein. Inputs to the Cocomo model are summarized in table 11 and the results predicted by the Cocomo model are depicted in table 12.

The Cocomo model computes estimates for cost, manpower, and schedule requirements for the following phases of software development by the contractor (only):

- o Preliminary design
- o Detailed design
- o Code and unit test
- o Integration and test

It does not include estimates for activities such as requirements analysis, formal demonstrations, and project administration.

The Cocomo Model predicts that JAMPS software development in Ada can be accomplished in 12.9 to 16.4 months; figure 5 indicates that this same work effort (Tasks 15-19, 21-25, 31-32) will be accomplished in 15 months.

The Cocomo Model predicts that software development will require 170 to 272 man months, depending on which set of assumptions is made. These predictions correspond with 151 mm (optimistic) and 344 mm (pessimistic) estimates computed via Method 1, and can be compared with the 180 actual man-months expended during the JAMPS implementation in "C".

The level of staffing predicted by the Cocomo Model (especially for the case involving pessimistic assumptions) indicates that a fairly large mainframe computer capable of supporting as many as 15 consoles will be needed for Ada software development.

The costs per delivered source statement, as computed by Method 1 and 2, are not directly comparable because the Cocomo Model does not consider such things as management reserve, contract monitoring expenses, etc. The cost estimates per source statement computed by Method 1, $110.74 and $179.46 for the optimistic and pessimistic cases, are less than the ESD average ($200.00/source statement); this is to be expected because 300 series procurement regulations are assumed.

Table 11

Inputs to Cocomo Model[15]

Development mode: organic, semidetached, embedded
Delivered source lines of code: 23,939 and 35,222 for optimistic and pessimistic cases, respectively
Preliminary design costs: $6200/mm
Detailed design costs: $6200/mm
Code and unit test costs: $6200/mm
Integration and test costs: $6200/mm

Cost Driver	Low	Low	Nominal	High	Very High	Extra High
RELY required software reliability	0.75	0.88	1.00	1.15	1.40	
DATA database size		0.94	1.00	1.08	1.16	
CPLX product complexity	0.70	0.85	1.00	1.15	1.30	1.65
TIME execution time constraint			1.00	1.11	1.30	1.66
STOR main storage constraint			1.00	1.06	1.21	1.56
VIRT virtual machine volatility		0.87	1.00	1.15	1.30	
TURN computer turnaround time		0.87	1.00	1.07	1.15	
ACAP analyst capability	1.46	1.19	1.00	0.86	0.71	
AEXP applications experience	1.29	1.13	1.00	0.91	0.82	
PCAP programmer capability	1.42	1.17	1.00	0.86	0.70	
VEXP virtual machine experience	1.21	1.10	1.00	0.90		
LEXP programming language experience	1.14	1.07	1.00	0.95		
MODP use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
TOOL use of software tools	1.24	1.10	1.00	0.91	0.83	
SCED required development schedule	1.23	1.08	1.00	1.04	1.10	

NOTE: Underlines are used to indicate the inputs selected. Numerical values are cost driver multiplication factors used in the model.

Table 12

Results from Cocomo Model

Optimistic Case

Phase	Man-Months	Cost (K$)	Schedule Months	Staff
Preliminary Design	20.1	124.8	3.0	6.8
Detailed Design (DD)	33.5	207.4	5.2	16.7
Code & Unit Test	52.8	322.2	included	in DD
Integration & Test	64.1	397.2	4.8	13.4
	170.4	1,056.6	12.9	

Productivity: 140.5 source statements/mm
Unit Cost: $44.14/Delivered Source Statements

Pessimistic Case

Preliminary Design	32.0	198.4	3.5	9.1
Detailed Design (DD)	52.4	325.1	5.8	23.2
Code & Unit Test	81.7	506.6	included	in DD
Integration & Test	106.0	657.3	5.7	18.5
Total	272.2	1,687.3	15.0	

Productivity: 129.4 source statements/mm
Unit Cost: 47.91/source statement

The most likely cost to duplicate JAMPS software in Ada, given the constraints provided herein, is the average of the optimistic and pessimistic projections computed via Method 1 (i.e., $4.5M).

SECTION 7

ADVANTAGES AND DISADVANTAGES OF REIMPLEMENTING JAMPS SOFTWARE IN ADA

7.1 ADVANTAGES

The principal benefits of undertaking an effort to recode JAMPS software in Ada are the following:

- The use of Ada will promote JAMPS software reusability by lessening the cost to incorporate JAMPS programs in other systems.
- The use of Ada is expected to decrease JAMPS life cycle costs because of improvements in software maintainability.
- ESD will benefit from the experience of an acquisition program involving the use of Ada.

7.2 DISADVANTAGES

The principal disadvantages in recoding in Ada are as follows:

- The Air Force has already invested 20 man-years in the JAMPS software written in "C" and most of this investment will be discarded if the software is rewritten in Ada.
- Air Force program offices which desire to use JAMPS software written in Ada will be held up three years while waiting for the Ada programming tools to mature and recoding of JAMPS software in Ada to take place.
- There are risks associated with using Ada at this time (e.g., programmer productivity with Ada is unknown, ESD has no experience with Ada in C3I applications).
- In the short run, code written in Ada is liable to be less reliable than the proven "C" software in JAMPS.

LIST OF REFERENCES

1. W. Kealy and K. Pigott, "Data Table Source File Description - Data Table Maintenance Manual," ESD-TR-82-125, Electronic Systems Division, AFSC, Hanscom AFB, MA (February 1982) A118476.

2. Telesoft-Ada sales literature, Telesoft, San Diego, California.

3. G. Moulton, "The ISCS-Ada Compiler," Irvine Computer Sciences Corporation.

4. R. DeLauer, "DoD Directive 5000.31, Computer Programming Language Policy," 10 June 1983 (draft).

5. J. Buxton, Department of Defense Requirements for Ada Programming Support Environments "STONEMAN" DoD, February 1982.

6. U.S. DoD, Reference Manual for the Ada Programming Language, MIL-STD-1815A, 17 February 1983.

7. J. Gilbreath and G. Gilbreath, "Eratosthenes Revisited, Once More Through the Sieve," Byte Magazine, January 1963, pp. 285-326.

8. Telesoft-Singer Librascope News Release, "Singer Uses Telesoft-Ada for Army, NATO, C3 Update, and It Works," 13 September 1983.

9. J. C. D. Nissen et al., "Ada-Europe Guidelines for the Portability of Ada Programs," National Physics Laboratory, NPL Report DNACS 52/81, November 1981.

10. H. Conn et al., "Ada Capability Study, Final Report," General Dynamics Report, DAAK80-81-C-0108, 30 June 1982.

11. H. Conn et al., "Ada Design Case Study, Ada Integrated Methodology," General Dynamics Report, Contract No. DAAK80-81-C-0108, 28 June 1982.

12. R. Wolverton, "The Cost of Developing Large Scale Software," COMPSAC 77 Tutorial Session.

13. Source EDP, 1983 Computer Salary Survey and Career Planning Guide.

LIST OF REFERENCES (Concluded)

14. LabTek Corporation WICAT hardware and Telesoft software price lists.

15. B. Boehm, Software Engineering Economics, Prentice Hall 1981, Englewood Cliffs, New Jersey.

APPENDIX A

A REPRESENTATIVE EXAMPLE OF JAMPS CODE REWRITTEN IN ADA

MITRE personnel at the Langley site selected a representative example of existing JAMPS code to be rewritten in Ada. This example was extracted from one of the off-line functions and is depicted in Attachment 1. Its purpose is to convert a file consisting of ASCII characters and numbers into a file consisting of ASCII characters, binary data, and an index. The following material is borrowed from D. J. Criscione.

During the conversion, some fundamental choices were made which affect the number of lines of Ada source code produced. Little regard was given to producing code which takes fullest advantage of the features offered by Ada. Instead, the resultant Ada code closely resembles the "C" code that was used as a model. Aside from necessary syntactic changes, only a few small sections were redesigned to accommodate language constructs which differ between "C" and Ada. As a result, the lines of executable code remain roughly the same. However, the total number of lines of Ada code is much larger than that of "C", because the Ada compiler requires explicit data declarations for describing the information contained in records (read in from disk) whereas the "C" compiler does not. The "C" listing sometimes contains several local variable declarations in a single line where Ada encourages the definition of a single data declaration per line, and this tends to distort the results shown below:

Category	Lines of "C"	Lines of Ada
Program Overhead	5	14
Constants used by the compiler	49	49
Specification of data types and data structures	23	35
Interface to UNIX files (code and specification)	3	22
Local variable declarations	7	35
Executable code	134	125
TOTAL	231	280
Brackets	-31	
Continuation Statements	-33	-36
Complete Source Statements	167	244

Due to the significant number of missing features in the Telesoft compiler used in this experiment, it was not possible to compile (without error diagnostics) the 280 lines of Ada source code referred to above. As a consequence, in numerous places, Ada code appears as comments in the listing wherever it failed to compile properly (see Attachment 2). Circumventing the compiler deficiencies, only 224 complete Ada source statements were actually compiled, resulting in 5,566 bytes of object code or 24.8 bytes per source statement.

Attachment 1: Source code listing for "C" code.
Attachment 2: Source code listing for Ada code.

Attachment 1

Representative Example Coded in "C"

```
/+++ defines +++ Data Base Defines - 07/07/83 */

#define     MAXMSGS          150     /* max messages in dir */
#define     MAXKDSS          600     /* max keyword in dir */
#define     MAXDFIS         1500     /* max dfis in dir */
#define     KDFIS             50     /* max dfi's in a kds table */
#define     MSTATES          100     /* max states per line (msg tables) */
#define     KSTATES          100     /* max states per line (kds tables) */
#define     DSTATES          100     /* max states per line (dfi tables) */
#define     MSTRMAX           62     /* max length of str for man flds */
#define     MMANFLDS           3     /* max mandatory fields in a msg */
#define     SHRTLEN            9     /* max length of short msg name */
#define     LONGLEN           20     /* max length of long msg name */
#define     KDSLEN             9     /* max character length of kds's */
#define     FDFHLEN           25     /* max character length of fdfh's */
#define     MAXVALL           35     /* max characters for validation
                                        MED template code area */
#define     DCHAINS           30     /* max E types for a single chain */
#define     DTABLE           200     /* max entries in a table macro */
#define     DDUIS            100     /* max duis for a single dfi */
#define     DTABSTR           40     /* max characters in a table entry */
#define     KRSTATE           50     /* Max number of states in a row */
#define     MSGINDEX           1
#define     KDSINDEX           2
#define     MSG                3
#define     KDS                4
#define     DFI                5
#define     FDFH               6
#define     INIT               1     /* initialize data tables mode */
#define     UPDATE             2     /* update existing data tables */
#define     DBGON              1     /* flag */
#define     DBGOFF             0     /* flag */
#define     MFILENM           14     /* maximum characters in file name */
#define     TEMPFILE "ibld.tmp"      /* temporary output file */
#define     STRFILE "ibld.str"       /* temporary table string file */
#define     CHN                1
#define     COMNOTAB           2
#define     COMTAB             3
#define     DIFNOTAB           4
#define     DIFTAB             5
#define     ALTCHN             6
#define     DIF                7
#define     C                  1
#define     E                  2
#define     FIX                1
#define     VAR                2
#define     UNKNOWN            0     /* unknown used when error detected */
#define     NA                -1
```

```
struct code
  {
     int cdfitype;          /* 1=C or 2=E for dfi type */
     int cdfi;              /* integer dfi number */
     int cdui;              /* integer dui number */
     int crange;            /* 1=integer range, 2=floating point range */
                            /* 3=octal range, -1=no range */
     float cfhigh;          /* value of upper boundary (float) */
     float cflow;           /* value of lower boundary (float) */
     long clhigh;           /* value of upper boundary (long) */
     long cllow;            /* value of lower boundary (long) */
     int cmin;              /* minimum code length of data item codes */
     int cmax;              /* maximum code length of data item codes */
     int clngth;            /* length in bytes of data item codes */
     int ccnt;              /* number of codes */
     long lcnext;           /* location of next dui */
  };
```

```
#include <stdio.h>
#include <ctype.h>
#include "defines"
#include "helpstruct"

#define  NEWDFI     1      /* Value returned from, and defined in lookahead */
#define  DUINAME    2      /* Value returned from, and defined in lookahead */
#define  LIT        3      /* Value returned from, and defined in lookahead */
#define  CODELINE   4      /* Value returned from, and defined in lookahead */
#define  RANGE      5      /* Value returned from, and defined in lookahead */
#define  OCT        6      /* Value returned from, and defined in lookahead */
#define  FLOAT      7      /* Value returned from, and defined in lookahead */
#define  ENDFILE    8      /* Value returned from, and defined in lookahead */
#define  MAXLINE    80     /* Maximum size buffer for line read             */

#define  INTEGER    1      /* Integer range indicator        */
#define  FLTPT      2      /* Floating Point range indicator */
#define  OCTRNGE    3      /* Octal range indicator          */

FILE *pin, *pout, *preadout, *fopen();
struct code_unde, coderead, *pcode, *pcread;
char sline[MAXLINE], scline[MAXLINE], sonecode[MAXLINE];
long lcurpos, lcurtmp, lcurloc, ltmpoff, ldfi, ftell();
int  linecnt, pos2, sizemin, sizemax;

main(argc,argv)
int argc;
char *argv[];


    int codelen, chk, s, loop1, loop2, pos1;
    int readchk = 0;
    char c;
    char *pline, *pdfiline, *pduiline, *pcline, *ponecode;
    char sdfiline[MAXLINE], sduiline[MAXLINE];


                    /*** GRAND OPENING ***/

    if (argc < 3)
    {
        fprintf(stderr,"*** ERROR *** an input and output file must ");
        fprintf(stderr,"be named\n");
        exit(1);
    }

    if ((pin = fopen(*++argv,"r")) == NULL)
    {
        fprintf(stderr, *** ERROR *** unable to open %s for read\n",*argv);
        exit(1);
    }

    if ((pout = fopen(*++argv,"w")) == NULL)
    {
        fprintf(stderr,"*** ERROR *** unable to open %s for write\n",*argv);
        exit(1);
    }
```

```
    if ((preadout = fopen(*argv,"r")) == NULL)
    {
        fprintf(stderr,"*** ERROR *** unable to open %s for read\n",*argv);
        exit(1);
    }

                    /*** INITIALIZATION CODE ***/

    pcode = &code;
    pcread = &coderead;

    /* Set cursor location to its initial position after the index */
    lcurloc = sizeof(long) * MAXDFIS;

    /* Set the position of pout (output file pointer) */
    fseek(pout,lcurloc,0);

    /* Initialize the line count of number of lines read */
    linecnt = 0;


                    /*** BEGIN PROCESSING ***/

#def DEBUG
    fprintf(stdout,"Processing has begun!\n");
#dif

    /* Initialize space for first read. */
    pline = sline;

    /* Get the first line of the file.  This loop will process the DFI's */
    /* and the inner loop will process the codes for each DFI. */

    while (readchk = fgets(pline,MAXLINE,pin))
    {
        if (readchk == EOF)
            preof();

    /* Call the initializing routine. */
        initial();

    /* Initialize the strings that are being used. */
        pdfiline = sdfiline;
        pduiline = sduiline;
        ponecode = sonecode;
        pciline = sciline;

    /* Find where the input file pointer is positioned and increment the */
    /* line count. */
        lcurpos = ftell(pin);
        ++linecnt;

    /* Since we have already checked this line, we know we are working with */
    /* a new DFI.  We can start extracting the information that we want. */
        if (*pline == 'E')
            pcode->cdfitype = E;
        else if (*pline == 'C')
            pcode->cdfitype = C;
        else
        {
            fprintf(stderr,"*** ERROR *** DFI type must be C or E. ");
            fprintf(stderr,"line %d\n",linecnt);
```

```
        exit(1);
    }
/* Convert the DFI number into an integer by substring and an ascii to */
/* integer conversion. */
    substr(pline,pdfiline,1,4);
    sdfiline[4] = '\0';

/* Before we do the conversion, we must replace the leading blank (if */
/* we have one) with zero. */
    if (sdfiline[0] == ' ')
        sdfiline[0] = '0';

    pcode->cdfi = patoi(pdfiline);

    if (pcode->cdfi < 0)
    {
        fprintf(stderr,"*** ERROR *** nonnumeric DFI number %d",
            pcode->cdfi);
        fprintf(stderr,", line %d\n",linecnt);
    }

    if (pcode->cdfi > MAXDFIS)
    {
        fprintf(stderr,"*** ERROR *** DFI number out of bounds = %d",
            pcode->cdfi);
        fprintf(stderr,", line %d\n", linecnt);
    }

/* Convert the DUI number into an integer by substring and an ascii to */
/* integer conversion. */
    substr(pline,pduiline,5,3);
    sduiline[3] = '\0';

    pcode->cdui = patoi(pduiline);

    if (pcode->cdui < 0)
    {
        fprintf(stderr,"*** ERROR *** nonnumeric DUI number %d",
            pcode->cdui);
        fprintf(stderr,", line %d\n", linecnt);
    }

/* Multiply the DFI number by 4 to get the proper offset for the */
/* index. */
    ldfi = 4 * (pcode->cdfi);

/* Clear any buffered information. */
    fflush(pout);
    fflush(preadout);

/* preadout = pointer to output file used for reading. */
/* pout = pointer to output file used for writing. */

/* Save the position of lcurloc. If there are codes or a range, we */
/* will want to enter this position in the index entry. */
    lcurtmp = lcurloc;

/* Seek to this position plus the size of the structure. */
    fseek(pout,(lcurloc + sizeof(*pcode)),0);
```

```
/* Look ahead to see if we are ready to process the codes.  Switch */
/* on the value returned. */
    while ((loop2 = lookahead(0)) != ENDFILE)
    {
        switch (loop2)

        {
            case NEWDFI:
                lcurloc = ftell(pout);
                prstruc(0);
                                        /* Return to outside loop   */
                break;                  /* to process next DFI.     */

            case DUINAME:
/* When the DUI name is continued on more than one line, the lookahead */
/* routine will adjusts the file pointer and increments the line count. */
/* Continue in loop looking ahead. */
                break;

            case LIT:
/* A LITERAL has been found.  We want to zero out any values that may */
/* have changed before discovering the LITERAL.  We search for the */
/* next DFI.  We are then prepared to look ahead, see the next DFI. */
/* and then we will write the structure info to file. */
                initial();
                search();
                prstruc(1);
                break;

            case CODELINE:
/* We are on a line that contains codes and a LITERAL and a range */
/* has not been found. We get the code, check to see if it is continued */
/* on the next line and write the code to file. */
                codelen = 0;

                pos1 = pos2;
                pcline = getcode(pline);

                while (((chk = lookahead(0)) == CODELINE) &&
                    (pos1 < pos2))
                {
                    strmove(pcline,ponecode);
                    pcline = concat(ponecode);
                }
                ++(pcode->ccnt);

/* When the first code is found, set the minimum and maximum code */
/* equal to the length of that code.  Otherwise compare the size of the */
/* code to see if it exceeds the size of the maximum or minimum code */
/* length, and act accordingly. */
                if (pcode->ccnt == 1)
                {
                    pcode->cmin = size(pcline);
                    pcode->cmax = size(pcline);
                }
                else
                {
                    if ((sizemin = size(pcline)) < pcode->cmin)
                        pcode->cmin = sizemin;
```

```
                    if ((sizemax = size(pcline)) > pcode->cmax)
                        pcode->cmax = sizemax;
                }

                codelen = size(pcline) + 1;
                pcode->clngth += codelen;

#ifdef DEBUG
                fprintf(stdout,"%s\n",pcline);
#endif

    /* Write the code to the file. */
                fwrite(pcline,sizeof(char),codelen,pout);

                break;

            case RANGE:
     * We have found a left parentheses followed by a non-alphabetic. We are*/
    /* ready to process a range. */
                range();
                break;

            default:
                fprintf(stderr,"*** ERROR *** Inapplicable return");
                fprintf(stderr," value = %d", loop2);
                fprintf(stderr,", line %d\n",linecnt);
                exit(1);
                break;
        }

        if ((loop2 == NEWDFI) || (loop2 == LIT))
            break;

    }
    if (loop2 == ENDFILE)
    {
        prstruc(0);
        preof();
    }
}   /* close inner loop */
/* end program*/
```

Attachment 2

Representative Example Coded in Ada

```
--  This package implements the insert 'DEFINES' used by the C implementation
--  of the JAMPS DFIDUI parser.
--
package DEFINES is

  MAXMSGS: constant integer := 150;
  MAXKDSS: constant integer := 600;
  MAXDFIS: constant integer := 1500;
  KDFIS:   constant integer := 50;
  MSTATES: constant integer := 100;
  KSTATES: constant integer := 100;
  DSTATES: constant integer := 100;
  MSTRMAX: constant integer := 62;
  MMANFLDS: constant integer := 3;
  SHRTLEN: constant integer :=5;
  LONGLEN: constant integer :=20;
  KDSLEN: constant integer :=9;
  FDFHLEN: constant integer :=25;
  MAXVAL1: constant integer :=35;
  DCHAINS: constant integer :=30;
  DTABLE: constant integer :=200;
  DDUIS: constant integer :=100;
  DTABSTR: constant integer :=40;
  KRSTATE: constant integer :=50;
  MSGINDEX: constant integer :=1;
  KDSINDEX: constant integer :=2;
  MSG: constant integer :=3;
  KDS: constant integer :=4;
  DFI: constant integer :=5;
  FDFH: constant integer :=6;
  INIT: constant integer :=1;
  UPDATE: constant integer :=2;
  DBGON: constant integer :=1;
  DBGOFF: constant integer :=0;
  MFILENM: constant integer :=14;
  TEMPFILE: constant string :="ibld.tmp";
  STRFILE: constant string :="ibld.str";
  CHN: constant integer :=1;
  COMNOTAB: constant integer :=2;
  COMTAB: constant integer :=3;
  DIFNOTAB: constant integer :=4;
  DIFTAB: constant integer :=5;
  ALTCHN: constant integer :=6;
  DIF: constant integer :=7;
  C: constant integer :=1;
  E: constant integer :=2;
  FIX: constant integer :=1;
  VAR: constant integer :=2;
  UNKNOWN: constant integer :=0;
  NA: constant integer :=-1;
end DEFINES;
--
```

```
with text_io,direct_io,JAMPS_STUBS,JAMPS_INPUT,JAMPS_OUTPUT,DEFINES;
use text_io;
use integer_io;
use defines;
--
--
package DFI_DUI_PARSER is
--

  -- The next line is necessary since Ada will not allow the
  -- usage of undefined types in type deffinition.
  type code;

  -- Some type deffinitions to be used in the structure 'CODE'.
  type access_code is access code;

  -- NOTE: Type code is used internally, but must be changed into
  --       a stream of characters via unchecked conversion to
  --       accomodate the character oriented C implementation.
  type code is
    record
      CDFITYPE:  integer;        -- This should be a user defined type
                                 -- since the only legitimate values are
                                 -- 1 and 2.
      CDFI:      integer;
      CDUI:      integer;        -- CDFI and CDUI should probably be user
                                 -- defined types, but I'm not sure what
                                 -- ranges would be legitimate.
      CRANGE:    integer;        -- This should also be a user defined
                                 -- type with values 'integer',
                                 -- 'floating_point', 'octal', and
                                 -- 'no_range'.
      CFHIGH:    float;
      CFLOW:     float;
      CLHIGH:    float;
      CLLOW:     float;
      CMIN:      integer;
      CMAX:      integer;
      CLNGTH:    integer;
      CCNT:      integer;
      LCNEXT:    access_code;
    end record;
```

```
  -- Another non-implemented feature.  Record representation clauses
  -- are not yet supported.  This means that the record structure here
  -- may not match the equivilant C structure.
  --for' code use
  --record at mod 8;
  WORD : constant integer:=4;  -- Assume 4 bytes per word.
  --   CDFITYPE    at 0*WORD range 0 .. 31;
  --   CDFI        at 1*WORD range 0 .. 31;
  --   CDUI        at 2*WORD range 0 .. 31;
  --   CRANGE      at  3*WORD range 0 .. 31;
  --   CFHIGH      at  4*WORD range 0 .. 63;
  --   CFLOW       at  6*WORD range 0 .. 63;
  --   CLHIGH      at  8*WORD range 0 .. 63;
  --   CLLOW       at 10*WORD range 0 .. 63;
  --   CMIN        at 12*WORD range 0 .. 32;
  --   CMAX        at 13*WORD range 0 .. 32;
  --   CLNGTH      at 14*WORD range 0 .. 32;
  --   CCNT        at 15*WORD range 0 .. 32;
  --   LCNEXT      at 16*WORD range 0 .. 63;
  --end record;


-- C allows unconstrained strings, Ada does not.  To accomodate the
-- existing interface a large string will be used, with a software check
-- for the delimiter used by C.
  procedure MAIN(ARGC : integer;           -- number of files
                 ARGV : string);           -- file names

  type WHICH_FILE_ERROR is (INPUT, OUTPUT);

end DFI_DUI_PARSER;


package body DFI_DUI_PARSER is

  procedure MAIN(ARGC : integer;
                 ARGV : STRING) is
    PIN          : long_integer;    -- SHOULD BE FILETYPE
    POUT         : long_integer;    -- SHOULD BE FILETYPE
    PIN_MODE     : constant JAMPS_INPUT.FILE_MODE := JAMPS_INPUT.IN_FILE;
    POUT_MODE    : constant JAMPS_OUTPUT.FILE_MODE := JAMPS_OUTPUT.IN_OUT_FILE;
    FILE_BEING_OPENED: WHICH_FILE_ERROR;
    SLINE        : JAMPS_INPUT.INPUT_LINE;
    SCLINE       : JAMPS_INPUT.INPUT_LINE;
    SONECODE     : JAMPS_INPUT.INPUT_LINE;
    PLINE        : JAMPS_INPUT.ACCESS_INPUT_LINE;
    PDFILINE     : JAMPS_INPUT.ACCESS_INPUT_LINE;
    PDUILINE     : JAMPS_INPUT.ACCESS_INPUT_LINE;
    PONECODE     : string(1 .. 80);
    PCLINE       : string(1 .. 80);
    STRING_END   : integer;
    STRING_START: integer;
    LINECNT      : integer;
    PCODE        : ACCESS_CODE;
    PCREAD       : ACCESS_CODE;
    LDFI         : long_integer;          --FILE_INDEX
    LCURPOS      : long_integer;          --FILE_INDEX
    LCURTMP      : long_integer;          --FILE_INDEX
    LCURLOC      : long_integer;          --FILE_INDEX
    LTMPOFF      : long_integer;          --FILE_INDEX
```

```
    FTEL          : long_integer;          --FILE_INDEX
    POS2          : integer;
    POS1          : integer;
    CODELEN       : integer;
    SIZEMIN       : integer;
    SIZEMAX       : integer;
    LOOP2         : JAMPS_STUBS.LOOKAHEAD_VALUE;
    TEMP_LONG:      long_integer;          -- used in integer to long integer
                                           -- conversion.
    TEMP_INT      : integer;
    OUTBUF        : character;
    COUNT         : integer;
    INPUT_INDEX   : integer;
  -- ********************* Start of Executable Code ****************************
    begin
      if ARGC < 3 then
        put_line("*** ERROR *** an input and output file must");
        put_line("be named");
        return; -- BAD_FILE_COUNT
      end if;

--    The delimiters used by the C interface for unconstrained strings
--    are scanned for here.  Might be better to do it in the JAMPS_INPUT
--    package.
      STRING_END:=1;
      STRING_START:=1;
      while ARGV(STRING_END .. STRING_END+1) /= "/n" loop
        STRING_END:=STRING_END+1;
      end loop;
      FILE_BEING_OPENED := INPUT;
      JAMPS_INPUT.open(PIN,PIN_MODE,ARGV(STRING_START .. STRING_END),"");
      STRING_END:=STRING_END+2;
      STRING_START:=STRING_END;
      while ARGV(STRING_END .. STRING_END+1) /= "/n" loop
        STRING_END:=STRING_END+1;
      end loop;
      FILE_BEING_OPENED := OUTPUT;
      JAMPS_OUTPUT.open(POUT,POUT_MODE,ARGV(STRING_START .. STRING_END),"");

--    At this point, the C code executes a separate OPEN to allow
--    reading from the output file.  This does not seem necessary
--    since the Ada DIRECT_IO package allows a file to be opened
--    for both reading and writing.
--
--                  *** INITIALIZATION CODE ***
      PCODE := new code;
      PCREAD:= new code;
      LCURLOC:=WORD*MAXDFIS;
      JAMPS_OUTPUT.set_index(POUT,LCURLOC);
      LINECNT := 0;
      PLINE := new JAMPS_INPUT.INPUT_LINE;
```

```
--  Get the first line of the file.  This loop will process the DFI's
--
MAIN_LOOP:
      loop
-- UNIX files are streams of characters, with logical records delimited
-- by a single delimiter character specified as '/n'.  For this estimate
-- I am assuming that the implementation of SEQUENTIAL_IO will return
-- the string as logical records with the '/n' stripped off.  The
-- logic which assumes simple stream input with no logical record
-- pre-processing is included if such support is not available.
--       for INPUT_INDEX in 1 .. SLINE'last loop
--         JAMPS_INPUT.read(PIN,INCHAR);
--         exit when INCHAR = '/n' ;
--         SLINE(INPUT_INDEX .. INPUT_INDEX) :=
--              unchecked_conversion(INCHAR);
--       end loop;
        JAMPS_INPUT.read(PIN,SLINE);
-- Check for terminating condition: End of input file.
        if JAMPS_INPUT.END_OF_FILE(PIN) then
        JAMPS_STUBS.Preof;
          put_line('Exiting procedure MAIN');
          return;  -- NORMAL_TERMINATION;
        end if;
-- Call the initialization routine
        JAMPS_STUBS.INITIAL;
-- Initialize the strings that are being used.
        PDFILINE := new JAMPS_INPUT.INPUT_LINE;
        PDUILINE := new JAMPS_INPUT.INPUT_LINE;
--      PONECODE := null;
--      PCLINE   := null;
-- Set the index and increment the line count.
        LCURPOS := JAMPS_INPUT.INDEX(PIN);
        LINECNT := LINECNT + 1;
-- Start extracting the information that we want.
        if SLINE(1 .. 1) = 'E' then
          PCODE.CDFITYPE := 1;
        elsif SLINE(1 .. 1) = 'C' then
          PCODE.CDFITYPE := 2;
        else
          put_line(' *** ERROR *** DFI type must be C or E');
          put(linecnt);
          return;  -- BAD_DFI_TYPE
        end if;
-- Convert the ASCII DFI number to an interger.
--    Note: I decided to handle possible blanks right here since that
--          is the way it was done in the C code.  It seems more
--          appropriate to handle blanks in the conversion routine.
        if SLINE(2 .. 2) = ' ' then
          SLINE(2 .. 2) := '0';
        end if;
--      pcode.cdfi := JAMPS_STUBS.ASCII_INTEGER(
--                    unchecked_conversion(SLINE(2 .. 5)),4);
-- Check for valid DFI number.
        if pcode.cdfi < 0 then
          put_line(' *** ERROR *** nonnumeric DFI number');
          put(pcode.cdfi);
          put_line(' line number');
          put(linecnt);
        end if;
```

```
if pcode.cdfi > MAXDFIS then
          put_line(' *** ERROR *** DFI number out of bounds =');
          put(pcode.cdfi);
          put_line(' line number');
          put(linecnt);
        end if;
-- Convert the DUI number.
        if SLINE(6 .. 6) = ' ' then
          SLINE(6 .. 6) := '0';
        end if;
--      pcode.cdui := JAMPS_STUBS.ASCII_INTEGER(
--                    unchecked_conversion(SLINE(6 .. 9)),4);
-- Check for valid DFI number.
        if pcode.cdui < 0 then
          put_line(' *** ERROR *** nonnumeric DUI number');
          put(pcode.cdui);
          put_line(' line number');
          put(linecnt);
        end if;
-- Multiply the DFI number by 4 to get the proper offset for the index.
--      ldfi := unchecked_conversion(4 * pcode.cdfi);
-- At this point, the C code flushes out the in core buffers.  There is
-- no equivilant function in the DIRECT_IO package.

-- Save the current position of lcurloc.
        lcurtmp := lcurloc;
-- Seek to this position in the output file.
        JAMPS_OUTPUT.set_index(POUT,lcurloc);

-- Look ahead to see if we are ready to process the codes.  Switch on
-- the value returned.
        LOOP2:=JAMPS_STUBS.LOOKAHEAD(0);
        INNER_LOOP: loop
          case LOOP2 is
            when JAMPS_STUBS.ENDFILE =>
              JAMPS_STUBS.prstruc(0);
              JAMPS_STUBS.proof;
              exit INNER_LOOP;
            when JAMPS_STUBS.NEWDFI =>
              LCURLOC:= JAMPS_OUTPUT.INDEX(POUT);
              JAMPS_STUBS.prstruc(0);
            when JAMPS_STUBS.DUINAME =>
              null;
            when JAMPS_STUBS.LIT =>
              JAMPS_STUBS.INITIAL;
              JAMPS_STUBS.SEARCH;
              JAMPS_STUBS.PRSTRUC(1);
            when JAMPS_STUBS.CODELINE =>
              CODELEN:=0;
              POS1:=POS2;
--            PCLINE:=unchecked_conversion(GETCODE(PLINE));
              while JAMPS_STUBS.LOOKAHEAD(0) = JAMPS_STUBS.CODELINE
                    and POS1 = POS2 loop
                PONECODE:=PCLINE;
                PCLINE:=JAMPS_STUBS.CONCAT(PONECODE);
              end loop;
              PCODE.CCNT:=PCODE.CCNT+1;
```

```
              if PCODE.CCNT = 1 then
                PCODE.CMIN:=PCLINE'last;
                PCODE.CMAX:=PCLINE'last;
                null;
              else
-- This section had to be re-written to accomodate the C assignment statement
-- within the IF statement.
--              SIZEMIN:=PCLINE'last;
                SIZEMAX:=SIZEMIN;
                if  SIZEMIN < PCODE.CMIN then
                  PCODE.CMIN:=SIZEMIN;
                end if;
                if  SIZEMAX > PCODE.CMAX then
                  PCODE.CMAX := SIZEMAX;
                end if;
              end if;
              CODELEN:=PCLINE'last +1;
              PCODE.CLNGTH := CODELEN +1;
-- Write the code to the file.
              for COUNT in 1 .. CODELEN loop
--              OUTBUF:=unchecked_conversion(PCLINE(COUNT .. COUNT));
                JAMPS_OUTPUT.WRITE(POUT,OUTBUF);
              end loop;
            when JAMPS_STUBS.RANGEY =>
              JAMPS_STUBS.RANGEX;
-- Not sure if these lines should be counted.  Considering the
-- type checking on LOOP2, the others case should never occur.
            when others =>
              put_line(' *** ERROR *** Inapplicable return: value=');
--            TEMP_INT:=unchecked_conversion(LOOP2);
              put(TEMP_INT);
              put_line(' line=');
              put(linecnt);
            end case;
-- At this point the C listing checks again for LOOP2 = NEWDFI or LIT
-- The lines are not included here (redundant) and should also be
-- excluded from the count of lines in the C listing.
          LOOP2:=JAMPS_STUBS.LOOKAHEAD(0);
        end loop INNER_LOOP;
      end loop MAIN_LOOP;
--
--      HANDLE EXCEPTIONS
--
      exception
        when NAME_ERROR | USE_ERROR =>
          if FILE_BEING_OPENED = INPUT then
            put_line("*** ERROR *** unable to open " & ARGV(STRING_START ..
                     STRING_END) & " for read");
            return; -- BAD_INPUT_FILE
          elsif FILE_BEING_OPENED = OUTPUT then
            put_line("*** ERROR *** unable to open " & ARGV(STRING_START ..
                     STRING_END) & " for write");
            return; -- BAD_OUTPUT_FILE
          end if;
    end;
  end DFI_DUI_PARSER;
```

END

FILMED

6-84

DTIC